Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Tvorba počítačové hry pomocí C# a Unity

Bakalářská práce

Autor: **Tomáš Doležel**

Vedoucí práce: Ing. Marek Pícka, Ph.D.

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Tomáš Doležel

Informatika

Název práce

Tvorba počítačové hry pomocí C# a Unity

Název anglicky

Creation of computer game in C# and Unity

Cíle práce

Cílem práce je vytvoření počítačové hry pomoci objektově orientovaného programování v jazyce C# za použití herního enginu Unity.
Bude se jednat o 2D hru pro jednoho až dva hráče ve které hráč ovládá vesmírnou loď s níž poráží nepřátele a tím postupuje úrovněmi. Po každé úrovní se zvýší obtížnost a hráč si bude moci za sesbírané peníze svou loď vylepšovat.

Metodika

Na začátku práce bude provedena analýza odborných zdrojů informací, která bude shrnuta v teoretické části práce. Následovat bude praktická část tvořena na základě těchto poznatků. V praktická části práce bude provedena analýza, návrh a implementace hry. Návrh bude vypracován pomocí modelovacího jazyka UML. Nakonec bude provedena implementace návrhu v multiplatformním herním enginu Unity programovacím jazykem C#.

Doporučený rozsah práce

30-40 stran

Klíčová slova

OOP, C#, Unity, programování, tvorba softwaru

Doporučené zdroje informací

CALABRESE, Dave. Unity 2D Game Development. 1.vyd. Packt Publishing Limited, 2014. ISBN 978-1-84969-256-4

EELES, Peter., CRIPPS, Peter. Architektura softwaru. Computer Press, 2011. ISBN: 978-80-251-3036-0

HANÁK, Ján. Objektovo orientované programovanie v jazyku C# 3.0. 1. vyd. Brno: Artax a.s., 2008. ISBN:978-80-87017-02-9

HANÁK, Ján. Praktické objektové programování v jazyce C# 4.0. 1. vyd. Brno: Artax a.s., 2009. ISBN:978-80-87017-07-4

Předběžný termín obhajoby

2018/19 LS – PEF

Vedoucí práce

Ing. Marek Pícka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 24. 1. 2019

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 1. 2019

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 15. 03. 2019

Čestné prohlášení

Prohašuji, že svou bakalářskou práci “Tvorba počítačové hry pomocí C# a Unity“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2019 _____

Poděkování

Chtěl bych poděkovat panu Ing. Marku Píckovi, Ph.D. za odborné vedení této bakalářské práce

Tvorba počítačové hry pomocí C# a Unity

Abstrakt

Tato práce se zabývá tvorbou počítačové hry v herním enginu Unity od společnosti Unity Technologies. Hra je naprogramována pomocí objektově orientovaného programování v programovacím jazyku C#. Kód byl napsán ve vývojovém prostředí VisualStudio od společnosti Microsoft.

Teoretická část práce popisuje teoretický základ nutný pro vytvoření hry. Napřed je zde popsán herní engine Unity. Následuje stručný popis programovacího jazyka C # a na konec jsou popsány základy objektově orientovaného programování. Praktická část práce se zabývá analýzou, návrhem a implementací. V části analýzy jsou stanové požadavky na hru a zkoumána jejich možná řešení. Následující část se zabývá návrhem jednotlivých třídy. K návrhu tříd byl použit modelovací jazyk UML. V poslední části jsou popsány vybrané části kódu.

Klíčová slova: OOP, C#, Unity, programování, tvorba softwaru

Creation of computer game in C# and Unity

Abstract

This work deals with the creation of a computer game in the Unity game engine by Unity Technologie. The game is programmed using object-oriented programming in C # programming language. The code was written in the integrated development environment VisualStudio by Microsoft.

The theoretical part describes the theoretical basis necessary for the creation of the game. First, the Unity game engine is described here. Then follows a brief description of the C # programming language and the basics of object-oriented programming. The practical part deals with analysis, design and implementation. In the analysis part, the requirements for the game are determined and their possible solutions are examined. The following section deals with the design of each class. The UML modeling language was used to design classes. The last part describes selected parts of the code.

Keywords: OOP, C#, Unity, programming, software creation

Obsah

Seznam obrázků

1 Úvod

Videoherní průmysl je jedním z nejrychleji rostoucích zábavních průmyslů 21. století. Svědčí o tom fakt, že za posledních několik desítek let se video hry vyvinuly z pár pohybujících se „kostiček“ na monitoru v něco takového, jako jsou dnešní hry vytvářené na technologii virtuální reality, kde si hráč téměř připadá jako by se ve hře skutečně nacházel.

Vytvořit vlastní počítačovou hru již není ani zdaleka tak obtížné, jako tomu bylo před několika lety. V dnešní době existují programy zaměřené přímo na vytváření her (tzv. herní engine), které poskytují funkce starající se například o vykreslování či simulaci fyziky, čímž vývojářům šetří spoustu práce s kódováním. Díky volně dostupným herním enginum si může vlastní jednoduchou hru vytvořit téměř kdokoliv a v některých případech i bez potřeby mít zkušenosti s programováním.

Tato bakalářská práce se věnuje vývoji počítačové hry v herním enginu Unity, vyvíjeným společností Unity Technologies a objektově orientovanému programování v jazyce C# ve vývojovém prostředí Visual Studio firmy Microsoft.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této práce je vytvořit počítačovou hru v herním enginu Unity za použití principů objektově orientovaného programování a programovacího jazyka C#. Bude se jednat o hru žánru space shooter pro jednoho až dva hráče, ve které hráč bude ovládat vesmírnou loď s níž se bude snažit porážet vlny nepřátel a tím postupovat do dalších, těžších úrovní. Za nasbírané peníze si hráč bude moci vylepšovat svou loď. Cílem této hry bude projít všemi úrovněmi. Hru nebude možné ukládat. Hra skončí, pokud poslední přeživší hráč přijde o všechny životy či se mu podaří všechny úrovně přežít. Hra bude obsahovat několik typů nepřátel, kteří se budou lišit podle typu jejich útoku.

2.2 Metodika

V teoretické části práce budou probrány technologie a postupy, které budou využité v části praktické. Praktická část bude rozdělena ne tři části: analýzu, návrh a implementaci. Na začátku bude provedena analýza, v které bude zkoumán řešený problém, definovány požadavky a identifikovány jednotlivé části systému. V další části dojde k návrhu tříd a vytvoření jednotlivých diagramů pomocí modelovacího jazyka UML. V části implementace budou naprogramovány jednotlivé třídy z diagramů tříd programovacím jazykem C# ve vývojovém prostředí Visual Studio. Hra jako taková bude vytvořena v herním enginu Unity. Jednotlivé grafické prvky budou vytvořeny v grafickém editoru GIMP.

3 Teoretická část

3.1 Unity

Unity je multiplatformní herní engine vyvinutý Unity Technologies k vývoji třírozměrných i dvourozměrných aplikací. Předností tohoto enginu je možnost vytvořit aplikaci až na 26 různých platforem, mezi které patří osobní počítače, herní konzole, mobilní telefony, chytré televize i platformy pro virtuální realitu.[1]

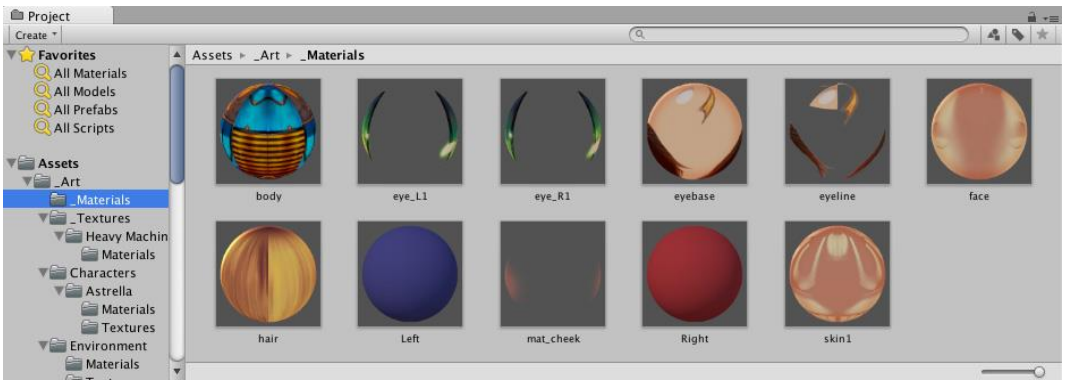
Unity nabízí několik typů licencí, od bezplatné osobní edice pro nezávislé vývojáře až po profesionální edici s měsíčními poplatky a prvky zaměřenými na výuku či profesionální podporou pro větší organizace.[2]

3.1.1 Grafické prostředí

Herní engine Unity je dostupný v Unity editoru, který je dostupný na platformách Windows a Mac. Editor má uživatelsky přívětivé grafické rozhraní a obsahuje nástroje pro animace, grafiku, optimalizaci, zvuky, 2D i 3D fyziku.[3] Editor je tvořen několika okny, nejpoužívanějšími jsou okna: *Project, Inspector, Scene, Game* a *Hierarchy.*

3.1.1.1 Project Window

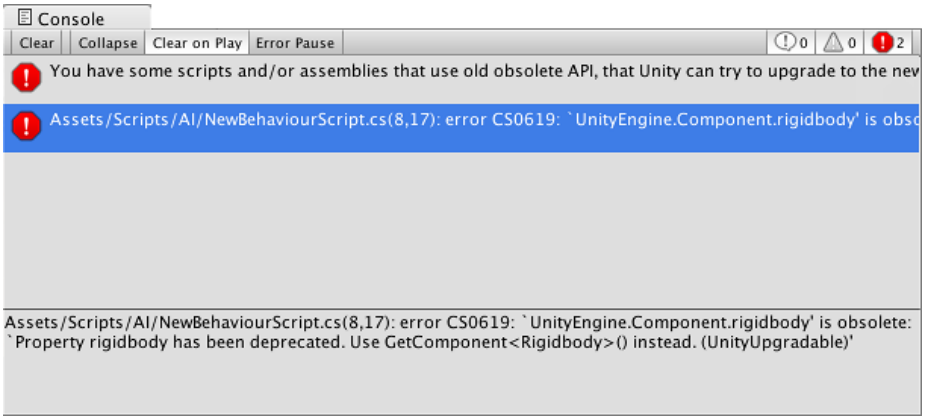
Toto okno slouží pro organizaci všech *assetů*. Asset je jakýkoliv soubor použitý v projektu. V levé části okna je zobrazena hierarchie složek. Výchozí složkou je složka *Assets*, která se pro zlepšení organizace rozšiřuje o uživatelsky definované podsložky. Podsložky se klasicky organizují podle typu assetů či herních objektů. V pravé části okna jsou zobrazeny jednotlivé assety a podsložky obsažené ve vybrané složce. Složky a assety zde lze vytvářet, odstraňovat, importovat i exportovat. Dalším prvek tohoto okna je i vyhledávací formulář, který umožňuje vyhledávat assetů dle jména či typu.[4]



Obrázek 1: The Project window [zdroj: 34]

3.1.1.2 Console Window

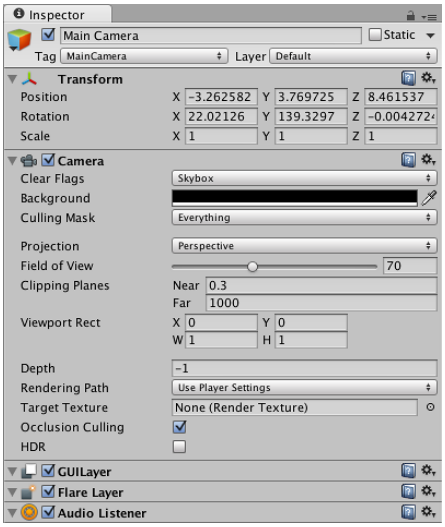
Editor unity obsahuje konzoli, v které se zobrazují všechny chybové hlášení a upozornění. Uživatel také může konzoli využívat k zobrazování vlastních zpráv, které si nastaví při psaní skriptů, to slouží například k usnadnění debuggování.



Obrázek 2: The Console window [zdroj: 35]

3.1.1.3 Inspector Window

Informace o vybraném herním objektu jsou zobrazeny v okně Inspector. Jsou zde všechny komponenty, z kterých se daný herní objekt skládá. Tyto komponenty zde lze přidávat, modifikovat i odebírat. Pokud má herní objekt přiřazen skript s veřejnými atributy, tak se tyto atributy zobrazí v tomto okně a je možné zde měnit jejich hodnoty.[5] V inspektoru je možné zobrazit a upravovat i privátní atributy. Toho lze dosáhnout přidáním označením privátního atributu ve skriptu klíčovým slovem *SerializedField*.[6] Toto je velmi užitečné, jelikož nám to umožňuje jednoduše pracovat s privátními poli bez toho, aby byli definované jako veřejné.

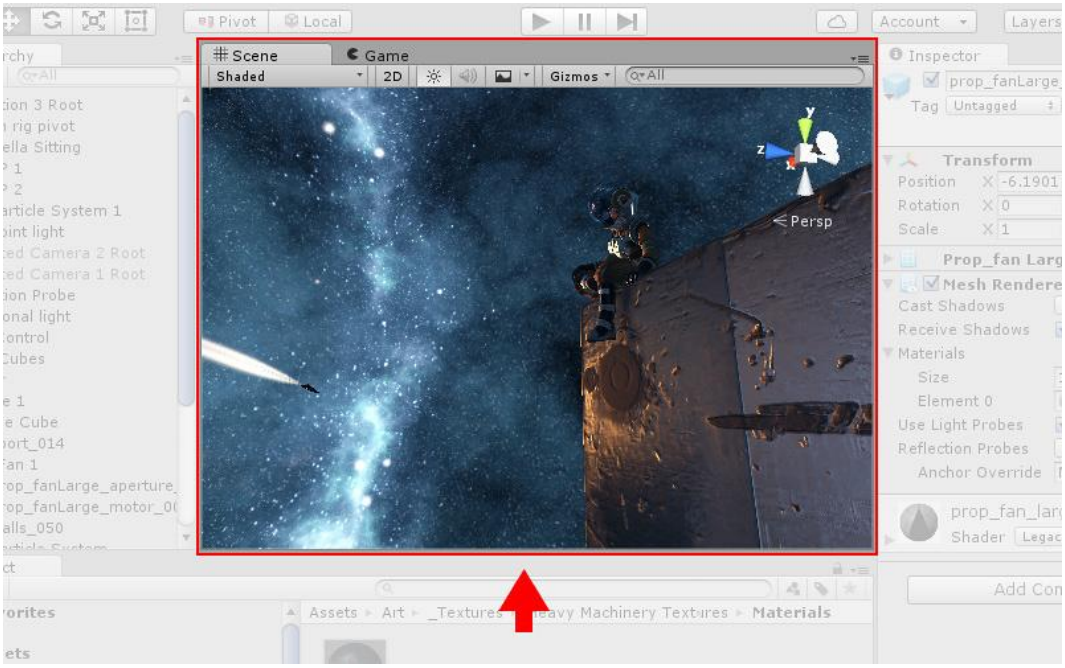


Obrázek 3: The Inspector window [zdroj: 36]

3.1.1.4 Scene View

V tomto okně jsou viditelné všechny herní objekty dané *scény*, jako jsou kamera, postavy, světla a všechny další typy objektů. Můžeme zde myší objekty vybírat a manipulovat s nimi.[7]

V horní části okna je panel, v kterém lze nastavit způsob vykreslování scény, přepínat mezi 2D a 3D zobrazením, vypínat a zapínat osvětlení, zvuky a efekty, nastavovat kameru a zobrazení a vyhledávat herní objekty v scéně.[7]

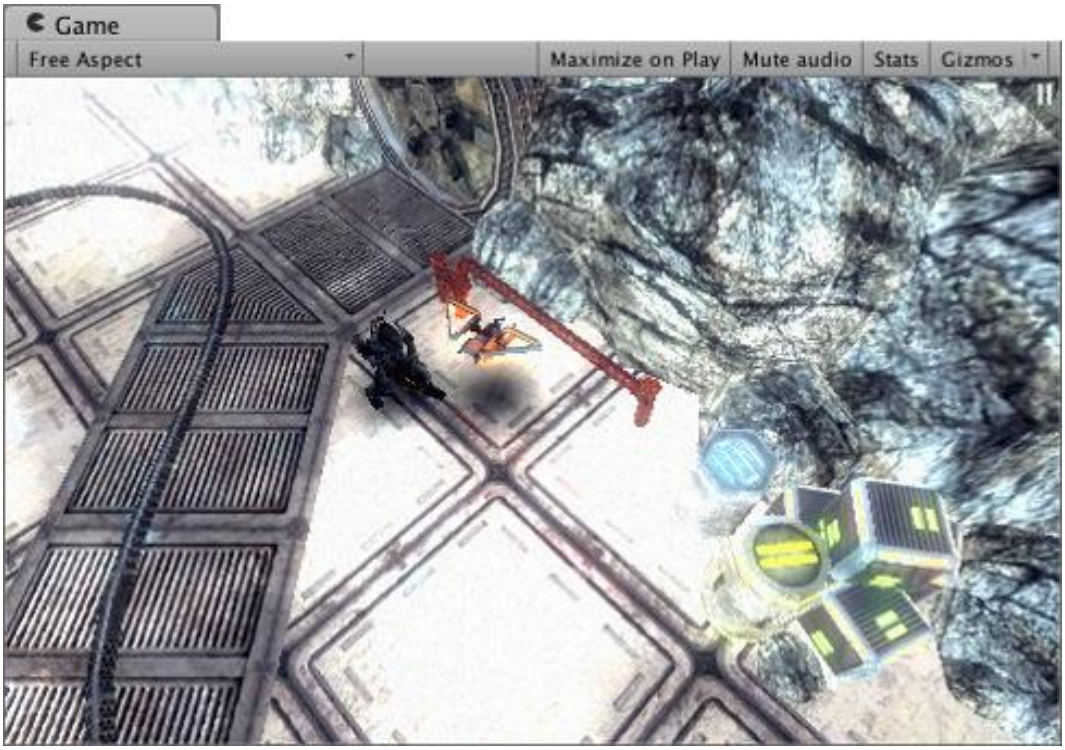


Obrázek 4: The Scene view [zdroj: 37]

Každý projekt se většinou skládá z několika scén. Scény tvoří jednotlivé úrovně hry i její menu. Například při spuštění hry se otevře scéna MainMenu z které, můžeme přejít například na scénu FirstLevel či Settings.

3.1.1.5 Game View

Toto okno poskytuje náhled na vybranou scénu, tak jak bude aktuálně ve hře vypadat. Pohled je dán kamerou, kterou lze nastavovat v okně Scene View. Hra může být testována a hrána díky tomuto oknu bez potřeby hru sestavit. Okno Game View může být při testování zvětšeno na celou obrazovku a také je možné zapnout statistiky, které ukazují vytížení procesoru a další údaje.[8]



Obrázek 5: The Game view [zdroj: 38]

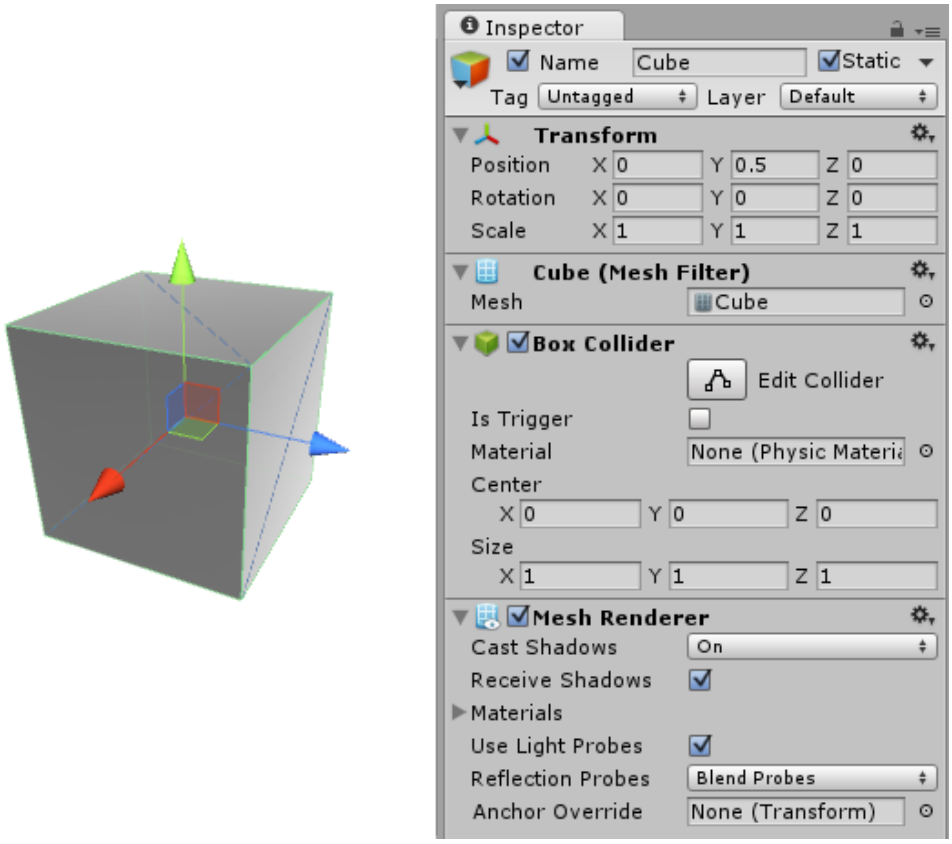
3.1.1.6 Hierarchy Window

Všechny herní objekty na dané scéně jsou zobrazeny v tomto okně. Vybráním herního objektu z hierarchie dojde k jeho otevření v okně Inspector a také k jeho zvýraznění ve scéně. Objekty mohou tvořit hierarchii objektů tvořenou předkem a potomkem. Tato hierarchie slouží ke sdílení komponenty *Transform* předka se svými potomky. K vytvoření potomka dojde přetažením jednoho objektu v okně Hierarchy na objekt, který má býti jeho předkem.[9]

Obrázek 6: The Hierarchy window [zdroj: 39]

3.1.2 Game Object

Objekty v Unity, ať už se jedná o postavu, předmět či speciální efekt, tak se nazývají *Game Object*. Každý game object, lze přeložit jako „herní objekt“, je tvořen z tzv. komponent, které stanovují jeho vlastnosti a chování, jako je jeho pozice ve scéně či způsob působení fyziky. Jednotlivé komponenty obsahují různé proměnné, jenž je možné upravovat během editace v inspektoru či za běhu programu skriptem. Dále si uvedeme několik nejběžnějších komponent využívaných v Unity.[10]



Obrázek 7: GameObject [zdroj: 40]

3.1.2.1 Transform

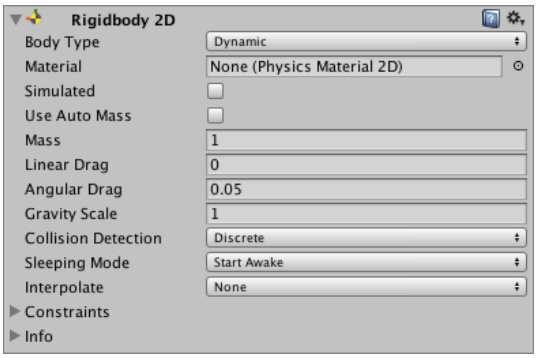
Komponentu Transform má v základu každý herní objekt a není možné jí odebrat. Tato komponenta představuje pozici, rotaci a měřítko herního objektu. Pozice herního objektu je dána podle hodnot jeho souřadnic X, Y a Z. Souřadnice Z je dostupná i pro dvourozměrné projekty a používá se zde k přesouvání objektů do popředí a pozadí.

Všichni potomci dědí tuto komponentu od svého předka. To má za následek, že jakákoliv změna pozice, rotace či měřítka předka změní tuto hodnotu i u potomka.[11]

3.1.2.2 Rigidbody

K tomu, aby na herní objekty působila fyzika slouží komponent *Rigidbody* a *Rigidbody 2D*. Rozdíl mezi těmito dvěma typy je ten, že Rigidbody2D se může za běhu programu pohybovat pouze ve dvou dimenzích, tedy po osách X a Y.

Způsob, kterým se herní objekty pohybují a interagují s ostatními závisí na položce *Body Type*. Body Type může být zvolen dynamický, kinematický či statický. Dynamické herní objekty jsou jediným typem, který může být mimo jiné ovlivňován i gravitací a jinými silami. S kinematickými herními objekty lze pohybovat pouze uživatelským vstupem. Na statické objekty nepůsobí, žádné síly a nemělo by s nimi být nikdy pohybováno.[12]



Obrázek 8: Rigidbody 2D [zdroj: 41]

3.1.2.3 Collider

To, že se herní objekty chovají během simulace jako pevná tělesa, která do sebe mohou narážet, místo toho, aby sebou jen procházeli, zajišťují komponenty zvané *collidery*. Těchto komponent existuje větší množství v závislosti na tvaru herního objektu a počtu dimenzí. Collidery také definují tvar herního objektu. Pokud herní objekt s tímto komponentem narazí do jiného, tak dojde zavolání funkce *OnCollisionEnter*. Poté je zde funkce *OnCollisionStay*, která je volána, dokud se herní objekty dotýkají. Jakmile dojde k přerušení dotyku herních objektů, tak se zavolá funkce *OnCollisionExit*. Těmito funkcemi lze pomocí skriptů pracovat a upravovat jejich chování.

Jakýkoliv collider může mít povolen atribut *IsTrigger* a je potom označován jako *Trigger Collider*. Trigger collider se při kontaktu s jiným colliderem nechová jako nepropustná zeď. Při kontaktu s jiným objektem dochází obdobně jako u klasických colliderů k volání funkcí *OnTriggerEnter*, *OnTriggerStay* a *OnTriggerExit*.

Podle nastavení komponenty Rigidbody na herním objektu, na kterém je collider připojen, lze rozdělit collidery na tři typy: *Static Collider*, *Rigidbody Collider* a *Kinematic Rigidbody Collider*. Za zmínku stojí statický collider, která jako jediný je připojen k hernímu objektu bez komponenty Rigidbody. Statický collider je vhodný například pro stěny a jiné herní objekty s kterými není možné pohybovat.[13]

3.1.2.4 Skripty

Unity nabízí velké množství komponentů, s kterými lze vytvořit komplexní herní objekty ale bez základní znalosti programování se tvůrce her neobejde. Skripty slouží k vytváření nových, uživatelsky definovaných komponentů. Tyto komponenty se používají k celé řadě věcí, například k přijímání uživatelského vstupu pro pohyb postavy, vytváření a úpravě herních objektů za běhu hry či k řízení událostní jako jsou kolize.

Výchozím jazykem pro vytváření skriptů v Unity je C#, a navíc je možné využívat další .NET jazyky, které umí kompilovat kompatibilní DLL (dynamické knihovny odkazů).[14]

V Unity každý nový skript automaticky dědí z třídy MonoBehaviour. MonoBehavior obsahuje mnoho užitečných prvků, jako jsou například funkce *Start*, *Update* a *FixedUpdate*. Funkce Start je volána při spuštění daného skriptu a používá se běžně k nastavení výchozích hodnot proměnných. Update je volán jednou za jeden snímek z každého skriptu, ve kterém je definován. Volání funkce Update je závisle na snímkové frekvenci a časový interval mezi jednotlivými volání, tedy není konstantní. Update se nejčastěji používá k příjmu uživatelského inputu či k implementaci jednoduchých časovačů. FixedUpdate se od Update liší v tom, že je nezávislí na snímkové frekvenci a časový interval mezi jednotlivými voláními je stejný. Všechny operace pracující s fyzickým enginem by se měli nacházet právě uvnitř FixedUpdate.[15]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class test : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {//… }

    // Update is called once per frame
    void Update()
    {//… }
}
```

3.1.2.5 Prefabs

Pokud chceme nějaký herní objekt vytvářet opakovaně v jedné či ve více scénách, tak místo toho, abychom daný objekt neustále kopírovali, můžeme využít tzv. prefabrikáty. Prefabrikát si můžeme představit jako hotový herní objekt, který již má přiřazeny všechny požadované komponenty, potomky a nastavené všechny atributy. S prefabrikáty pak můžeme jednoduše pracovat editoru i ve skriptech. Pokud chceme za běhu hry vytvářet nové objekty, tak bychom měli používat právě prefabrikátů. Typicky se prefabrikáty používají pro projektily či nepřátele. Prefabrikát lze vytvořit tím, že herní objekt z okna Hierarchy přesuneme do okna Project. Při změně prefabrikátu dojde ke změně ve všech instancích. To neznamená, že při úpravě některé z instanci dojde ke změnám i v prefabrikátu.[16]

3.1.2.6 Tagy a Vrstvy

K identifikaci objektů ve skriptech slouží mimo jiné i tzv. tagy. Pomocí tagů můžeme jednoduše rozlišovat a vyhledávat objekty ve skriptech. Tagy se běžně používají například při kolizích, kdy je potřeba zjistit o jaký typ objektu se jedná.[17]

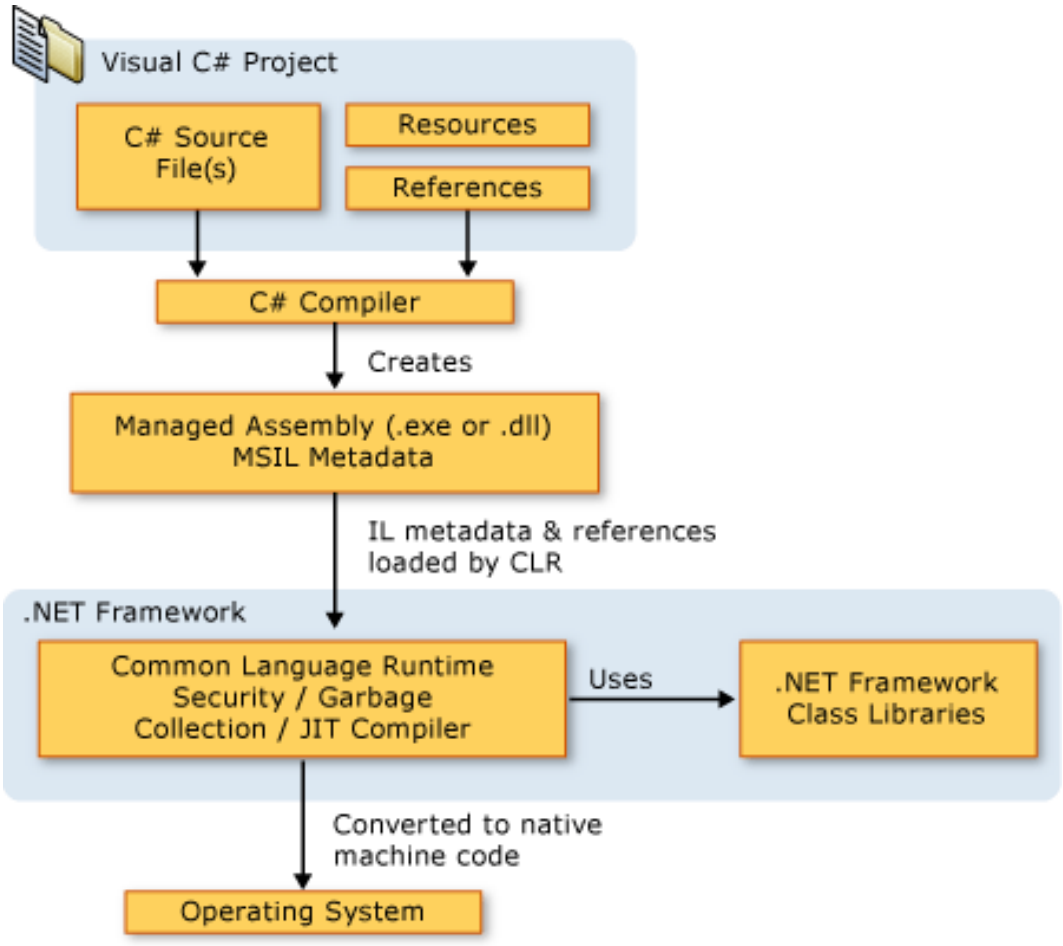
Vrstvy se používají k vytváření skupin objektů, která sdílí nějaké stejné charakteristiky. Běžně se používají k omezení procesů jako je například vykreslování. U objektů s rozdílnými vrstvami lze také nestavit ignorování kolizí. [18]

V okně Inspector lze tagy a vrstvy snadno vytvářet a přiřazovat k jednotlivým objektům.

3.2 Programovací jazyk C#

Jazyk C# je typově bezpečný, objektově orientovaný programovací jazyk od společnosti Microsoft. Syntaxí se tento jazyk podobá programovacím jazykům jako jsou Java, C a C++.

Programy napsané v C# běží na .NET frameworku, což je integrovaná součást operačního systému Windows, která obsahuje virtuální stroj nazvaný common language runtime (CLR) a skupinu knihoven tříd. Zdrojový kód je zkompilován do intermediate language (IL), který je poté spolu s ostatními zdroji uložen na disku ve spustitelném souboru nazývaném assembly. Při spuštění C# programu je assembly načten do CLR, jenž provede just in time (JIT) kompilaci k převedení IL kódu na strojový kód.[19]



Obrázek 9: Architektura .NET Framework [zdroj: 42]

3.3 Objektově orientované programování

Objektově orientované programování, je styl programování, který se snaží řešit daný problém modelováním reality. Zaměřuje se na použití *abstrakce* k definování *objektů* pro návrh a vytváření programů. Objekt může představovat jakoukoliv reálnou věc, pojem i proces, například auto, schůzku i počasí. Objekty navzájem komunikují a vykonávají požadované činnosti. Objektově orientovaný přístup bývá v řadě případů efektivnější než přístup procedurální, který je tvořen posloupností příkazů. Tento přístup se snaží umožnit vysokou znovupoužitelnou kódu, rozložit úlohu na menší části, skrýt detaily implementace před uživateli a přiblížit strukturu řešení reálnému světu.[20]

3.3.1 Objekt

Objekt je základní entitou objektově orientovaného sytému. Každý objekt je charakterizován svým *identifikátorem*, *vlastnostmi* a *chováním*. Identifikátor slouží k jednoznačnému určení objektu v systému. Vlastnosti objektu, též označované jako *atributy*, jsou proměnné sloužící k uložení hodnot popisující daný objekt. Hodnoty uložené v atributech definují vnitřní stav objektu. Chování objektu je reprezentováno jeho *metodami*. Metody slouží k přístupu k atributům, práci s nimi a také ke komunikaci s ostatními objekty v systému.[21]

Komunikace mezi objekty probíhá zasíláním a přijímáním zpráv podobným způsobem, jako si zprávy předávají lidé. To, jaké zprávy je objekt schopen přijmout a zpracovat je dáno jeho rozhraním. Po přijetí zprávy objekt spustí metodu, která je se zprávou spjatá, vykoná danou činnost a výsledek vrátí odesílateli.[22]

3.3.2 Abstrakce

Abstrakce představuje proces, při kterém zanedbáváme nepodstatné detaily a zaměřujeme se jen na ty podstatné. Toto je využíváno především při návrhu tříd, při kterém musíme vhodně definovat společné charakteristiky daných objektů, tak abychom vybrali pouze ty charakteristiky, které jsou pro námi řešený problém podstatné a tím se vyhnuli zbytečné komplexnosti systému.[23]

3.3.3 Zapouzdření a skrývání informací

Zapouzdření a skrývání informací jsou dva související pojmy. Zapouzdření představuje uzavření dat a metod do jedné jednotky, tedy objektu, a skrývání informací označuje ukrytí těchto charakteristik objektu před jeho okolím. Ukrytím charakteristik objektu, se vyvarujeme nechtěné modifikaci dat jiným objektem.[24]

Skrytí informací dosáhneme pomocí přístupových modifikátorů. Nejběžnějšími modifikátory přístupu jsou modifikátory *public*, *private* a *protected*, lze přeložit jako „veřejný“, „soukromý“ a „chráněný“. Člen označený jako veřejný může být zpřístupněn jakýmkoliv jiným objektem nebo částí programu. Oproti tomu soukromý člen mohou používat jen členy třídy v které se vyskytuje. Chráněný člen je podobný jako soukromý, ale mohou k němu přistupovat i členy odvozených tříd. Všechny atributy objektu by měli být definované jako soukromé či chráněné, aby k nim nešlo přistupovat z vnějšku objektu.

Přímý přístup k datům uloženým v těchto atributech by měli mít pouze tzv. *přístupové metody* („accessors“). Přístupové metody slouží k získání a k modifikaci dat uložených v soukromých atributech. K získání hodnoty atributu slouží metoda zvaná *getter* a k modifikaci této hodnoty metoda *setter*.[25]

```
//zkrácený zápis
string _myProperty { get; set; }
//úplný zápis
string _myProperty;
public string getMyProperty()
{return this._myProperty; }
public void setMyProperty(string value)
{this._myProperty = value; }
```

Všechny metody až na ty, které budou sloužit ke komunikaci s ostatními objekty, se definují také jako soukromé či chráněné. Množina těchto veřejných metod se označuje jako *rozhraní objektu*.[26]

3.3.4 Třída

Objekty se stejnými atributy a metodami patří do stejné skupiny objektů. Tato skupina stejných objektů se nazývá *třída*. Třídu si můžeme představit jako šablonu, podle níž jsou vytvářeny objekty. V třídě jsou definované atributy a metody, které budou všechny

objekty z této třídy vytvořeny obsahovat. Jakmile je definována třída, můžeme vytvářet libovolný počet objektů této třídy.

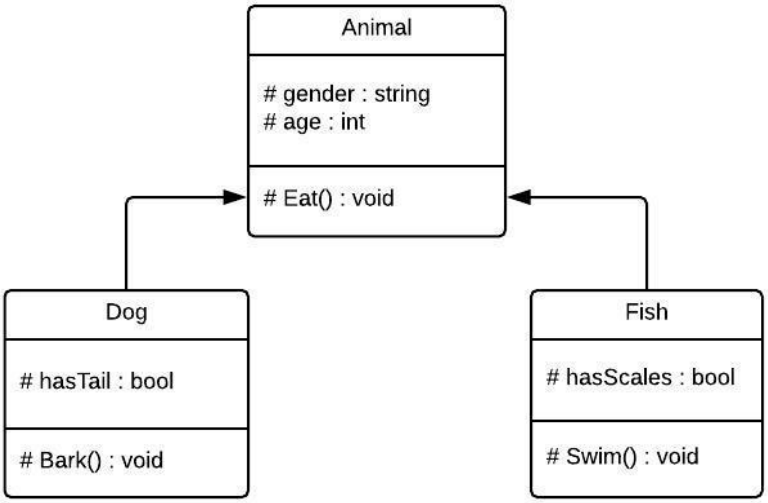
```
public class Car
{
	//atributy
	private string plateNumber;
	private int speed;
	//konstruktor
	public Car(string plateNumber, int speed){
		this.plateNumber = plateNumber;
		this.speed = speed;
	}
	//metody
	public void Move(){//…}
	public void Park(){//…}
	public void Accelerate(){//…}
}
```

Proces vytváření objektů se nazývá instancování a objekty takto vytvořené se nazývají *instance* třídy.[27] Každá instance třídy tedy sdílí stejné atributy a metody, ale liší se v hodnotách uložených v těchto atributech.

```
class TestCar
{
	static void Main()
	{
		// vytvoření instance
		var car1 = new Car("123abc", 10);
		var car2 = new Car("234bcd", 20);
	}
}
```

3.3.5 Dědičnost

Dědičnost je proces, díky kterému mohou objekty přebírat vlastnosti a chování objektů jiných. Při dědičnosti nová třída přebírá charakteristiky třídy, z které je vytvořena. Třída, z které je nová třída odvozena se označuje jako *nadtřída* či *předek* a třída odvozená se označuje jako *podtřída* či *potomek*. Podtřída přijímá všechny charakteristiky své nadtřídy, a navíc může být rozšířena o nové atributy i metody. Z jedné třídy lze vytvořit libovolný počet podtříd. Pokud třída dědí jen z jedné třídy jedná se o tzv. jednoduchou dědičnost. Naopak, když třída dědí z více, než jenom z jedné třídy hovoříme o tzv. vícenásobné dědičnosti. Z dědičnosti vyplívá, že podtřída může dělat to samé co nadtřída plus něco navíc, jedná se tedy o konkrétnější případ nadtřídy. [28]



Obrázek 10: Dědičnost

3.3.6 Polymorfismus

Využitím *polymorfismu* můžeme pracovat se skupinou objektů určité hierarchie objektů skrze jednotné rozhraní. Polymorfismus umožňuje, aby rozdílné objekty mohli reagovat na stejnou zprávu rozdílným způsobem.[29] Dva způsoby dosažení polymorfismu jsou *přetěžování* a *přepisování* metod. Přetěžování metod znamená, že jedna třída může mít několik metod s rozdílnou implementací ale stejným názvem. To, která z metod bude přijetím zprávy spuštěna závisí na typu a počtu předaných parametrů.

```
public class Pretezovani
{
      public string Add(string a, string b)
      {return a + b;}
      public int Add(int a, int b)
      {return a + b;}
      public int Add(int a, int b, int c)
      {return a + b + c;}
}
```

Přepisování metod souvisí s dědičností. Občas je potřeba aby zděděná metoda měla rozdílnou implementaci, než má v nadtřídě, a právě k tomu slouží přepisování metod.[30]

```
      public class Shape
      {
            public virtual void Draw()// Virtuální metoda
            {//základní implementace metody}
      }
      class Circle : Shape
      {
            public override void Draw()//přepsání virtuální metody rodiče
            {//změněná implementace metody}
      }
      class Rectangle : Shape
      {
            public override void Draw()
            {//...}
      }
      class Program
      {
            static void Main(string[] args)
            {
                  var shapes = new List<Shape>//vytvoření objektů
                  {new Rectangle(),new Circle()};
                  foreach (var shape in shapes) //práce s objekty
                  {shape.Draw();}
            }

      }
```

3.3.7 Abstraktní třídy

Abstraktní třída je speciální typ třídy, z které nelze vytvářet instance, ale slouží pouze jako šablona pro vytváření dalších tříd. Abstraktní třídy mohou obsahovat abstraktní metody. Abstraktní metoda je taková metoda, která je pouze deklarovaná ale není implementována. Třídy dědící z abstraktní třídy musí implementovat všechny její abstraktní metody.[31] Abstraktní třídy se používají tam, kde máme skupinu objektů, které budou mít stejné chování, ale každý ho bude vykonávat trochu jinak.

```
abstract class ShapesClass //definice abstraktní třídy
{
    abstract public int Area();//definice abstraktní metody
}
//definice třídy dědící abstraktní třídu
class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {side = n;}
    //implementace (přepsání) abstraktní metody
    public override int Area()
    {return side * side;}
}
```

3.3.8 Rozhraní

Rozhraní se principem podobá abstraktní třídě, ale nejedná se o třídu. Udává, jaké veřejné metody musí mít třídy, které jej implementují. Narozdíl od abstraktní třídy, rozhraní obsahuje pouze deklarace názvů metod a jejich implementace je zcela přenechána na třídách, které toto rozhraní implementují. Rozhraní není třídou, proto je možné, aby jedna třída implementovala několik rozhraní najednou. Stejně jako u tříd, je možné, aby rozhraní dědilo z jiného rozhraní.[32]

```
public interface IFood // definice rozhraní
{
      //deklarace veřejné metody
      public void Prepare();
}
//definice třídy implementující rozhraní
public class Pizza : IFood
{
      //implementace metody rozhraní
      public void Prepare()
      {//...}
}
```

3.3.9 Skládání objektů

Kromě dědičnosti můžeme k vytváření nových objektů využít i tzv. *skládání objektů*. Skládání objektů umožňuje vytvářet objekty z jednodušších objektů. Tyto jednodušší objekty jsou uloženy v atributech takto vytvořeného objektu.[33]

4 Praktická část

4.1 Analýza

4.1.1 Sběr požadavků

Bude se jednat o dvourozměrnou počítačovou hru, kterou je možné hrát v jednom či ve dvou hráčích. V této hře každý hráč ovládá vesmírnou loď, s kterou se může pohybovat ve spodní části obrazu do stran a střílet projektily. Hráčův pohyb bude limitován bočními hranicemi a nebude možné těmito hranicemi procházet.

Hráč má za úkol úspěšně projít všemi úrovněmi. To se mu podaří, pokud jeho životy během hry neklesnou na nulu. V každé úrovni se vytvoří několik nepřátelských lodí, které poletí směrem dolu a při tom budou vystřelovat projektily. Když se nepřítel dostane za hráče, tak hráč ztratí část svých životů. O životy hráč přijde i v případě, že se srazí s nepřátelskou střelou či s nepřítelovou lodí. Hráč se tedy bude muset snažit co nejrychleji ničit nepřátele, a přitom se co nejlépe vyhýbat jejich střelám. Aby nebylo tak snadné hru vyhrát, hráč během hry nedostane žádnou možnost, jak si životy své lodi doplnit či navýšit.

Při zničení nepřátelské lodě bude existovat šance na to, že z nepřítele vypadnou peníze. Při nasbírání určitého množství peněz si hráč bude moci během hry vylepšovat svou loď. Vylepšovat půjde rychlost pohybu a rychlost střelby lodě.

Pokud se hráči podaří porazit všechny nepřátele, tak postoupí do další úrovně. Při postupu do další úrovně se hráčovi životy nedoplní a bude tedy pokračovat s tolika životy, kolik mu jích zůstalo z předchozí úrovně. Každá následující úroveň se oproti té předchozí stane výrazně obtížnější.

Během hry, by měl hráč vidět informace o své lodi, jako jsou životy a stupeň vylepšení lodi. Také bude by měla být zobrazena aktuální úroveň, v které se hráč nachází.

4.1.2 Analýza požadavků

4.1.2.1 Cíl hry

Oproti velké části podobných her není cílem této hry nasbírat co nejvyšší skóre, ale projít všemi úrovněmi. Jelikož sběr skóre ani není v požadavcích definován, tak nebude ve hře skóre implementováno. Po dokončení poslední úrovně hra končí a hráč se stává vítězem.

4.1.2.2 Úrovně

Hra se bude skládat z několika úrovní, které se mají lišit v obtížnosti. Na začátku každé úrovně dojde k vytvoření několika nepřátel v horní části obrazu, kteří budou klesat dolu a při tom vystřelovat projektily. Obtížnost úrovní může být dána více faktory. Mezi ně může patřit například počet nepřátel, jejich rychlost pohybu či střelby, síla jejich projektilů nebo jejich typ střel.

Systém úrovní může být vyřešen více způsoby. Jedním z nich je ten, že každá úroveň bude tvořena jednou předem vytvořenou scénou. Pokud by byl zvolen tento přístup, tak by se v každé scéně museli předem vytvořit nepřátelé, kteří by se při každém hraní úrovně chovali stejně, letěli ve stejném pořadí, nebo by bylo možné je v náhodném pořadí aktivovat, aby se hra nestala při opakovaném hraní monotónní. Druhým způsobem může být vytvoření třídy, která se bude starat o vytváření jednotlivých úrovní. V tomto případě by bylo využito náhodného generování úrovní dle předem zadaných parametrů, jako je například počet nepřátel.

4.1.2.3 Lodě

Hráčské i nepřátelské lodě budou obsahovat komponenty jako jsou *Rigidbody2D* a *Collider2D*. Každá loď, ať již hráčská či nepřátelská, se bude muset pohybovat, střílet a obdržovat poškození. K tomu lze přistoupit více způsoby. Jedním z nich je vytvořit abstraktní třídu *Ship*, která bude definovat společné vlastnosti a chování pro oba typy lodí. Z této třídy by byly odvozeny třídy pro hráčskou a nepřátelskou loď. Každá z těchto tříd by měla vlastní implementaci metod přizpůsobenou tomu, zda se jedná o hráčem či počítačem ovládanou loď. Dalším způsobem je využitím skládání objektů vytvořit samostatnou třídu pro každou z potřebných činností. Tímto způsobem by vznikli třídy pro zdraví, střelbu, pohyb a podobně. Pro ovládání těchto tříd by bylo potřeba další třídy, která by jim pouze předávala zprávy dle typu ovládání lodi. Tato třídy by byla abstraktní a byly by z ní odvozeny dvě třídy, kde jedna z nich by přijímala hráčský vstup a druhá by generovala vstup nepřítele.

4.1.2.3.1 Pohyb

Pohyb hráče má být umožněn pouze horizontálně. Hráčova loď bude umístěna ve spodní části obrazu, aby měl čas se vypořádat s nepřáteli, kteří budou klesat z horní části.

Jelikož se jedná o počítačovou hru, tak pohyb může být ovládán pomocí klávesnice, myši či jiného ovladače. Rychlost pohybu hráče bude dána stupněm jeho vylepšení.

Pohyb nepřátel je na rozdíl od pohybu hráče vertikální a není v požadavcích nijak jinak specifikován. Rychlost jeho pohybu může být tedy konstantní či se postupně zvyšovat či snižovat.

Veškeré pohybující se herní objekty budou muset mít přiřazenou komponentu *Rigidbody2D*.

4.1.2.3.2 Střelba

Jak hráč, tak i nepřítel má být schopen vystřelovat projektily. Toto může být implementováno stejným způsobem, jako ve hře Space Invaders, kde hráč střílí pomocí tlačítka určeného pro střelbu a nepřítel střílí v náhodných časových intervalech. Rychlost střelby hráče má záviset na úrovni vylepšení této vlastnosti. Při stisknutí tlačítka pro střelbu, tedy hráčova loď vystřelí pouze tehdy, pokud uběhla dostatečná doba od posledního výstřelu. Aby hráč nemusel neustále mačkat tlačítko pro střelbu, bylo by vhodné, pokud by tlačítko stačilo držet stisknuté a střelba se bude po daném časovém intervalu sama opakovat.

4.1.2.3.3 Projektily

Chování projektilů není v zadání nijak upřesněno. V některých hrách existuje i více typů projektilů, které mají různé chování. Rychlost nepřátelských střel by měla být větší, než jejich rychlost pohybu, aby nepřátelé při svém klesajícím pohybu nenechávali své střely za sebou. Projektily budou při nárazu do nepřátelské lodi způsobovat poškození a dojde k jejich zničení.

4.1.2.3.4 Peníze

Po zničení nepřítele má občas dojít k vypadnutí peněz. To, jestli z nepřítele při jeho zničení vypadnou peníze, může být rozhodnuto již při vytvoření nepřítele nebo naopak až při jeho zničení. Při vytvoření nepřítele to lze udělat například uložením instance peníze v poli třídy nepřátelské lodi. Při jeho zničení o vytvoření peníze může rozhodovat metoda, která při smrti nepřítele vygeneruje náhodné číslo, které porovná s danou šancí na vypadnutí peníze a podle výsledku bude peníz vytvořen, či ne.

4.1.2.3.5 Kolize

Hra jako taková bude založená na kolizích mezi objekty. Bude se muset rozlišovat, jaké typy objektů do sebe naráží, protože při kolizích mezi různými objekty má docházet k rozdílným událostem a některé kolize se musí ignorovat. Ze zadání vyplývá většina událostí, ke kterým při kolizích daných objektů dojde, ale například co se má stát při kolizi projektilů či lodí stejného typu není definováno. U projektilů můžeme vyřešit kolize tak, že při nárazu projektilu hráče do projektilu nepřítele dojde buď k ignorování kolize a projektily sebou pouze prolétnou, nebo dojde ke srážce projektilů a projektily se navzájem zničí. Kolize lodí stejného typu bude ignorována, protože náraz jednoho hráče do druhého je nevyhnutelný, když se oba dva hráči budou pohybovat ve stejné rovině. Také bude vhodné vyřešit, co se stane s projektily, penězi a nepřáteli, kteří opustí hráčem viditelný obraz, aby nedocházelo k jejich hromadění a tím k zatěžování počítače. Toto půjde vyřešit například neviditelnou stěnou, která bude tyto objekty ničit, když se jí dotknout, nebo časovačem, jenž po nějaké době tyto objekty sám zničí. Všechny tyto herní objekty budou muset obsahovat komponent typu *Collider2D*.

4.1.2.3.6 Vylepšování

Vylepšovat má jít rychlost střelby i pohybu. V zadání je stanoveno, že vylepšovat loď se má dát během hry. To může být provedeno více způsoby. Například pomocí nějaké vylepšovací stanice, před začátkem další úrovně, či přímo v průběhu úrovně jen zmáčknutím tlačítka pro vylepšení. Ve většině hrách, kde se dá nějakým způsobem vylepšovat některou z vlastností několikrát za sebou, při tom většinou stoupá nejen úroveň vlastnosti, ale i cena dalšího vylepšení.

4.2 Návrh

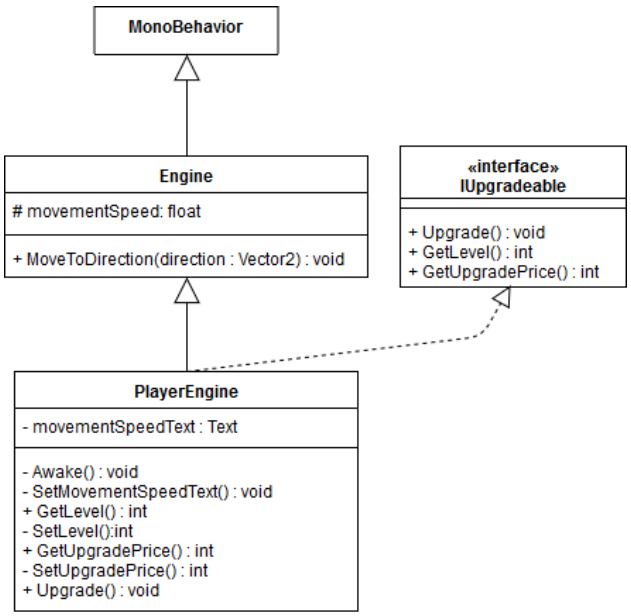
4.2.1 Lodě

Hráčské i nepřátelské lodě budou obsahovat komponenty *Transform, Rigidbody2D* a *PolygonCollider2D*. Každá loď bude mít motor, zbraň a životy. Každou z těchto částí bude reprezentovat skript, který si můžeme představit jako jednu třídu. Loď bude ovládána buď uživatelským vstupem či v případě nepřátelské lodě, náhodně generovaným vstupem. To bude mít na starost samostatná třída.

4.2.1.1 Pohyb

Třída *Engine* se bude starat o pohyb lodě. Bude odvozena z třídy *MonoBehaviour* a bude obsahovat pouze jeden atribut a jednu metodu. Atribut bude typu float a bude sloužit k uložení hodnoty rychlosti pohybu. Metoda se bude starat o pohybování lodi. Směr pohybu bude dán přijatým parametrem.

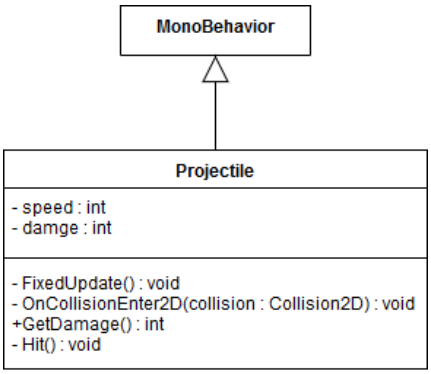
Z této třídy bude odvozena třída *PlayerEngine*, která bude implementovat rozhraní *IUpgradeable*, které bude obsahovat metody a vlastnosti pro vylepšovatelné objekty. Bude zde metoda *Awake* pro inicializaci výchozích hodnot vylepšení a textu. Mimo to bude tato třída obsahovat i odkaz na textové pole, v kterém budou zobrazeny informace o hráčově motoru, a metodu starající se o nastavení tohoto textu. K hráčské lodi poté bude přiřazen skript *PlayerEngine* a k nepřátelské skript *Engine*.



Obrázek 11: Třída Engine

4.2.1.2 Střelba

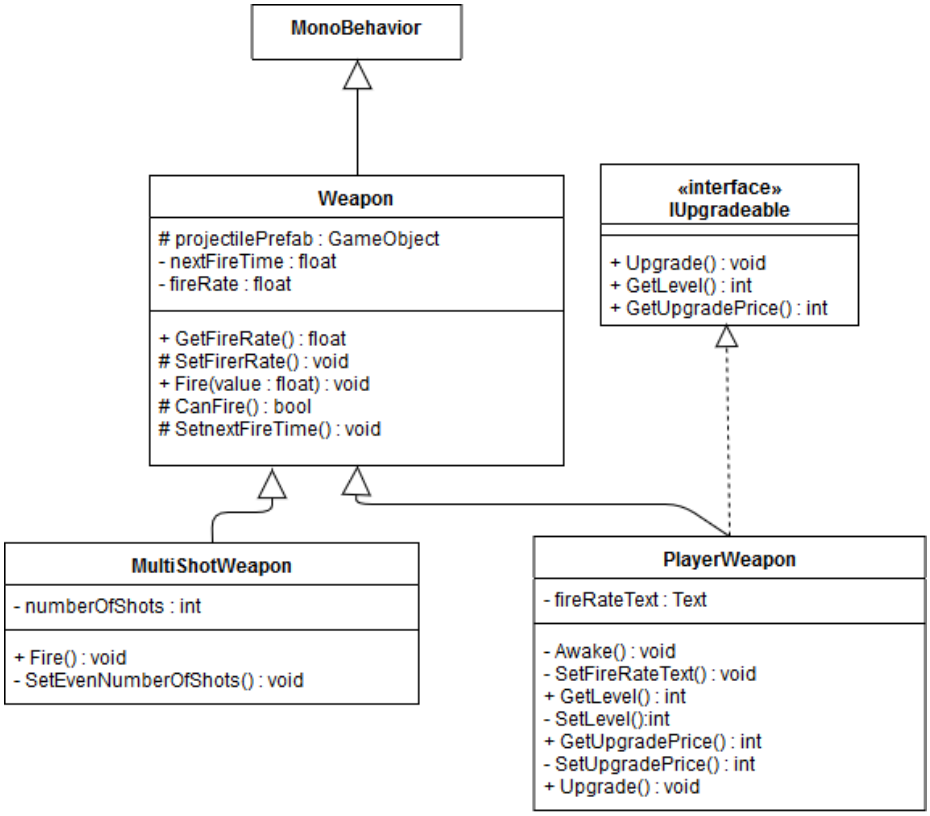
Herní objekt *Projectile* bude muset obsahovat komponenty typu *Collider2D* a *Rigidbody2D*. Bez nich by nebylo možné s ním pohybovat a kontrolovat kolize. Také bude obsahovat skript s třídou Projectile, která bude mít dva privátní atributy pro poškození a rychlost letu projektilu. Bude dědit z třídy *MonoBehaviour* a v metodě *FixedUpdate* se bude starat o pohyb projektilu. Dále bude mít veřejnou metodu, která bude vracet poškození a také jednu privátní metodu starající se o zničení projektilu. Metoda pro zničení projektilu bude zavolána, pokud dojde ke kolizi, a to pomocí metody *OnCollisionEnter2D* z třídy *MonoBehaviour*.



Obrázek 12: Třída Projectile

Skript *Weapon* obsahující stejnojmennou třídu se bude starat o vytváření projektilů. Tato třída bude opět dědit z třídy *MonoBehaviour*. Tělo třídy bude obsahovat odkaz na prefabrikát herního objektu *Projectile*, privátní atribut pro rychlost střelby a další privátní atribut pro dobu, než bude možné znovu vystřelit. Tato třída bude mít několik metod. Budou zde přístupové metody k atributu rychlosti střelby, metoda starající se o vytváření projektilů z prefabrikátu a metody spravující rychlost střelby.

Z třídy *Weapon* boudou dědit další třídy zbraní. První z nich bude třída *PlayerWeapon,* která podobně jako třída *PlayerEngine* implementuje rozhraní *IUpgradeable* a stará se o zobrazení informací o hráčově zbrani. Další třída dědící z třídy *Weapon* bude třída *MultiShotWeapon*, která se oproti *Weapon* bude lišit v počtu vystřelovaných projektilů.

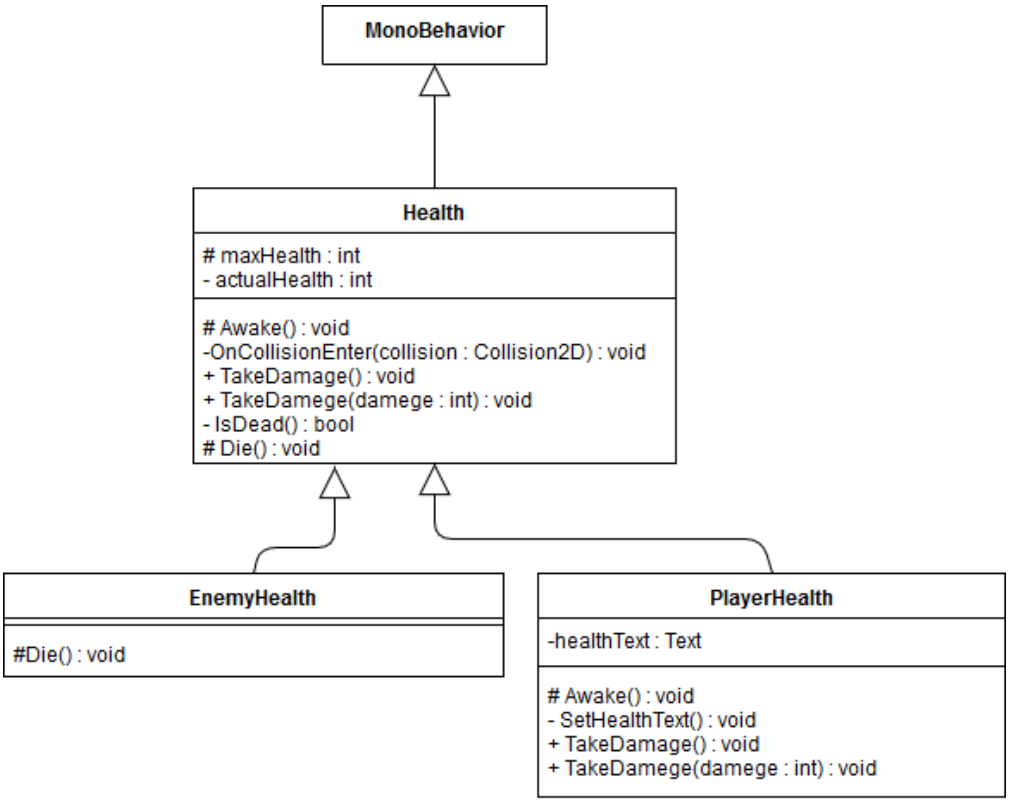


Obrázek 13: Třída Weapon

4.2.1.3 Zdraví

Třída *Health* bude abstraktní třída, která bude mít na starost přijímání poškození a v případě klesnutí životů na nulu i zničení lodě. Stejně jako předchozí třídy bude dědit z třídy *MonoBehaviour*. Třída *Health* bude obsahovat dva atributy. Jeden pro hodnotu maximálního počtu životů a druhý pro hodnotu aktuálního počtu životů. Metoda *Awake* se postará o inicializaci výchozích hodnot. Další metody budou kontrolovat, zda životy lodě klesly na nulu, udělovat poškození a vypořádávat se s kolizemi, které mají mít za následek ztrátu životů.

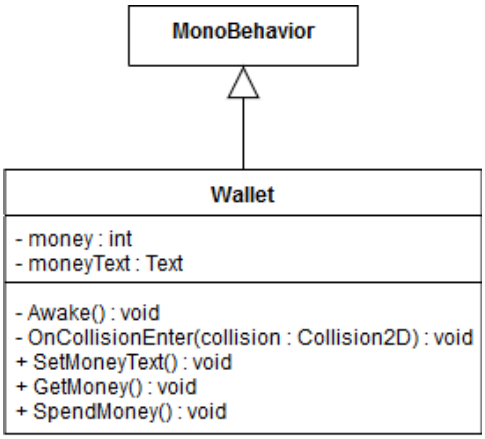
Z této abstraktní třídy budou odvozeny třídy *PlayerHealth* a *EnemyHealth*. Třída *PlayerHealth* se bude od výchozí třídy *Health* lišit podobně jako třídy PlayerEngine a *PlayerWeapon* v zobrazování údajů o hráčově zdraví během hry, a také v implementaci kolizí, které bude implementovat trochu jinak, než tomu bude v třídě *EnemyHealth*. Třída *EnemyHealth* se bude od třídy *Health* lišit pouze v kolizích.



Obrázek 14: Třída Health

4.2.1.4 Sběr a správa peněz

Skript *Wallet* je jediný z těchto skriptů, který bude mít pouze hráčská loď. Třída *Wallet* v sobě bude ukládat hodnotu peněz, které hráč posbírá během hry. Také se bude starat o jejich vypisování, utrácení a sbírání. Obsahuje tedy atribut k uložení sumy peněz a odkaz na textové pole. Opět tato třída bude dědit z třídy *MonoBehaviour*. V metodě *Awake* nastaví textové pole. Také bude obsahovat metodu *OnCollisionEnter2D*, která se bude starat o sběr peněz.



Obrázek 15: Třída Wallet

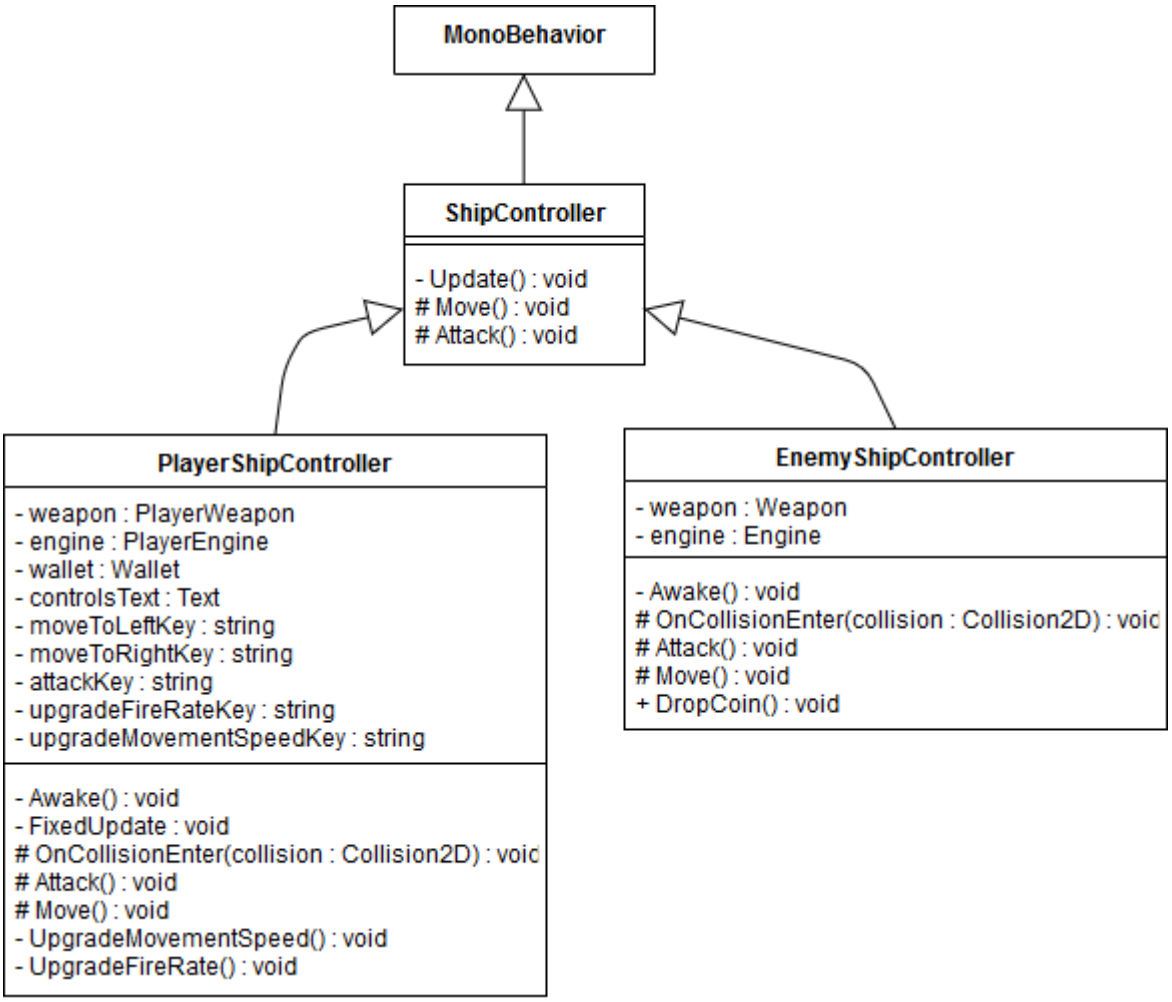
4.2.1.5 Ovládání lodi

Tato třída se bude starat o ovládání pohybu a střelby hráče i nepřítele. Bude se jednat o abstraktní třídu dědící z *MonoBehaviour*. V metodě *Update* bude volat dvě abstraktní metody. Jednu pro pohyb a druhou pro střelbu. Z této třídy budou odvozeny třídy *PlayerShipController* a *EnemyShipController*.

Třída *PlayerShipController* bude kontrolovat hráčem stisknutá tlačítka a pokud pro nějaké z těchto tlačítek bude mít nastavené nějaké chování, tak jej provede. V atributech bude mít uložené odkazy na jednotlivé části lodi, které jsou potřebné pro provedení daného úkonu. Dále bude mít v atributech uloženy kódy pro jednotlivá tlačítka, které budou po stisknutí spouštět danou metodu. Také zde bude atribut pro text zobrazující přiřazení tlačítek, aby hráč věděl, co jaké tlačítko dělá. V metodě *Awake* budou vyhledány potřebné komponenty lodě a nastaven výchozí text ovládání. Metoda *FixedUpdate* bude použita pro kontrolu stisknutí tlačítka pro vylepšení motoru a zbraně lodi. Dále zde budou implementovány abstraktní metody pro pohyb a střelbu z rodičovské třídy, ve kterých bude

docházet také ke kontrole stisknutí tlačítek. Nakonec tu budou dvě metody, které se budou starat o vylepšování motoru a zbraně.

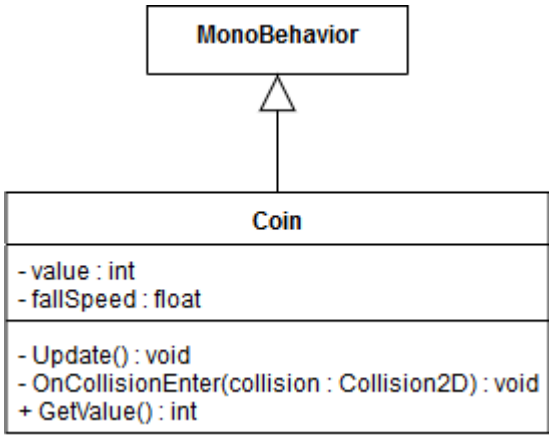
Třída *EnemyShipController* na rozdíl od *PlayerShipController* nebude kontrolovat zmáčknutí tlačítek na klávesnici, ale bude pomocí zadaných hodnot a náhodných hodnot generovat chování sama. Stejně jako v třídě k ovládání hráče bude mít v atributech odkazy na potřebné komponenty, které se v metodě *Awake* naleznou a uloží do těchto atributů. Dále bude mít atribut určující pravděpodobnost výstřelu a odkaz na prefabrikát. Dále zde budou implementovány abstraktní metody z rodičovské třídy. Nakonec zde bude metoda *OnCollisionEnter2D*, která bude kontrolovat, zda se nepřítel dostal za hráčovu lokaci.



Obrázek 16: Třída ShipController

4.2.2 Peníze

Peníze budou tvořeny herním objektem s komponentami *Rigidbody2D*, *Collider2D* a skriptem *Coin*. Třída Coin bude dědit z MonoBehaviour a bude obsahovat dva privátní atributy. První z nich bude obsahovat hodnotu peníze a druhý rychlost jakou bude peníz padat. Metoda *Update* bude mít na starost padání peníze. Také zde bude metoda *OnCollisionEnter2D*, která peníz při jeho nárazu do jiného objektu zničí. Nakonec tu bude přístupová metoda k hodnotě peníze, kterou použije třída *Wallet* ke zjištění, kolik peněz se má přidat an účet. Z herního objektu *Coin* se poté vytvoří prefabrikát.



Obrázek 17: Třída Coin

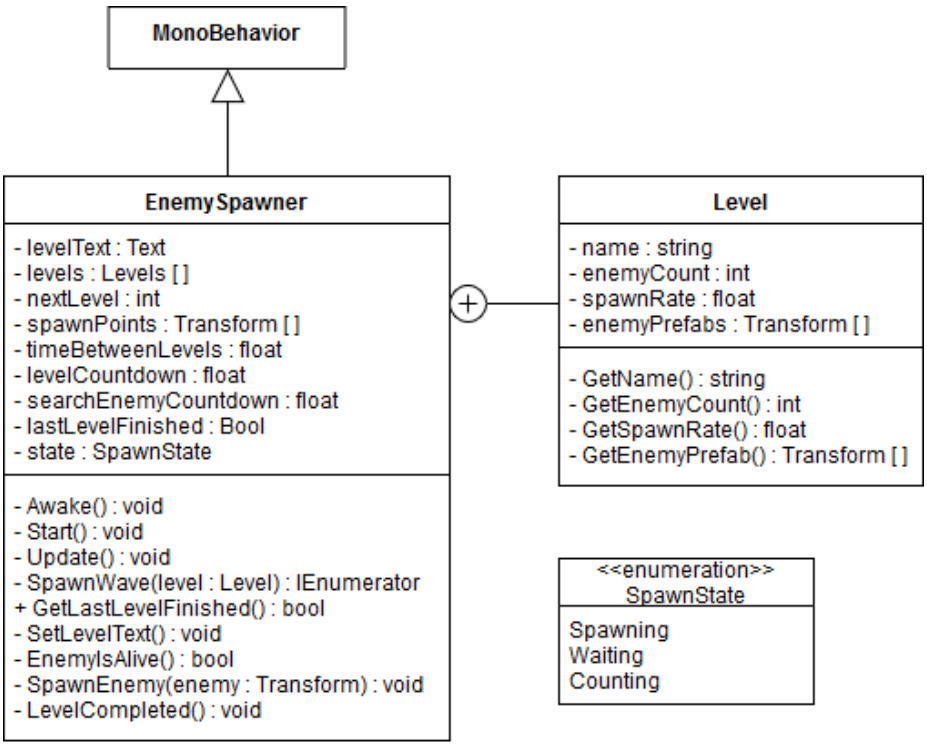
4.2.3 Úrovně

Všechny úrovně hry budou vytvořeny v jedné scéně pomocí třídy *EnemySpawner* a *Level*. Třída *Level* bude privátní vnořená třída v těle třídy *EnemySpawner*.

Třída *Level* bude obsahovat pouze atributy, které budou definovat jednotlivé úrovně hry. Všechny atributy této třídy budou privátní a každý z nich bude mít přístupovou metodu k získání hodnoty z atributu. Bude zde atribut pro název úrovně, počet nepřátel, rychlost vytváření nepřátel a pro pole, který bude obsahovat prefabrikáty všech nepřátel v dané úrovni. Žádné jiné metody tado třída obsahovat nebude.

Oproti třídě *Level* bude třída *EnemySpawner* výrazně větší. Úkolem této třídy bude postupně tvořit jednotlivé úrovně. To bude dělat tak, že pro každou úroveň vygenereju náhodné nepřátele z pole nepřátel daného úrovně na náhodně vybraných herních objektech, které budou sloužit pouze jako lokace, na kterých se může nepřítel objevit. Nepřátel udělá daný počet a mezi vytvořením nepřátel bude nechávat určitý čas. Po zničení všech nepřátel

počká třída danou dobu a poté se přesune na další úroveň. Třída bude obsahovat pole s jednotlivými úrovněmi. Dále tu bude pole se všemi tzv. *spawnPoints*, což jsou lokace herních objektů, které udávají místa, kde se mohou nepřátelé objevovat. Dalším atributem této třídy bude atribut držící index následující úrovně, atribut s prodlevou mezi jednotlivými úrovněmi, atribut sloužící jako počítadlo prodlevy mezi úrovněmi, atribut s doubou mezi kontrolováním počtu nepřátel a několik dalších. Ve třídě bude i výčtový typ pro jednotlivé stavy vytváření nepřátel, které se budou ukládat do privátního atributu. Tato třída opět dědí z třídy *MonoBeaviour*. V metodě *Awake* najde všechny *spawnPoints* a nastaví text oznamující aktuální název úrovně. Metoda *Update* bude řídit proces vytváření úrovní. Dále zde bude metoda pro vytváření úrovně, metoda pro kontrolu, zda již byla dokončena poslední úroveň, metoda sloužící k vytvoření nepřítele, metoda kontrolující, zda již byli všichni nepřátele poražení, metoda k nastavení atributů po dokoknčení úrovni a samozřejmě metoda pro nastavení textu zobrazujícího aktuální úroveň.

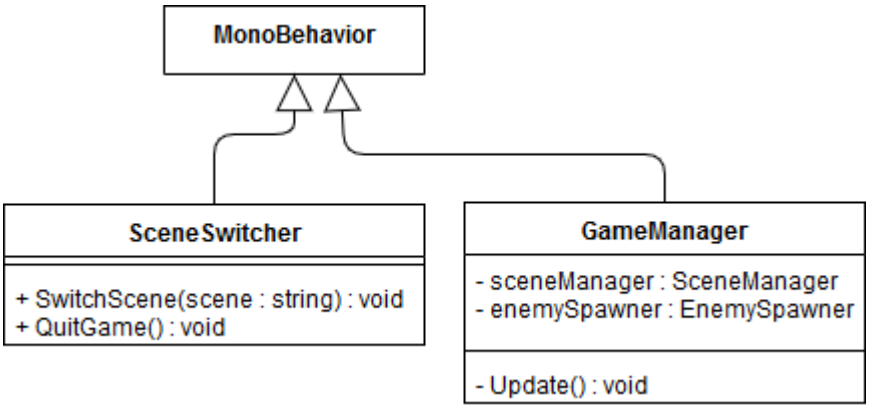


Obrázek 18: Třída EnemySpawner

4.2.4 Třídy SceneSwitcher a GameManager

Obě dvě tyto třídy budou dědít z třídy *MonoBehavior*. Třída *SceneSwitcher* se bude starat o přepínání mezi jednotlivými scénami. Bude obsahovat pouze dvě metody. Jedna z nich přepne aktuální scénu na scénu určenou přijatým parametrem a druhá vypne program. Obě tyto metody budou použity k ovládání tlačítek v menu hry.

GameManager bude třída starající se pouze o přepínání scén při konci hry. V privátních atributech bude mít uložené odkazy na objekty *SceneSwitcher* a *EnemySpawner*, které se v metodě *Awake* najdou a do těchto atributů uloží. Metoda Update bude kontrolovat, zda došlo k dokončení poslední úrovně či k poražení všech hráčů. Pokud jedna z těchto situací nastane, tak *SceneSwitcher* přepne na příslušnou scénu.



Obrázek 19: Třída GameManager a SceneSwitcher

4.2.5 Hranice hry

Hranice hry, tedy část scény, na které se budou jednotlivé herní objteky vyskystovat bude vyřešena pomocí čtyř herních objektů se statickými collidery. Dva tyto objekty budou představovat pravou a levou zeď, která bude limitovat hráčův pohyb. Pohyb nepřátel, projektilů a peněz bude limitovat horní a spodní zdí. Tyto dvě zdi budou umístěny trochu za viditelný obraz, aby hráč neviděl, jak objekty do nich narážející zmizí.

4.3 Implementace

4.3.1 Přepínání scén a ukončování hry

Přepínání mezi scénami a ukončování hry zajišťuje skript a stejnojmenná třída *SceneSwitcher*. Pro práci se scénami musí *SceneSwitcher* používat jmenný prostor *UnityEngine.SceneManagement.* Tato třída je velmi jednoduchá a obsahuje pouze dvě metody.

Metoda *SwitchScene* je typu void a přijímá jeden parametr typu string, který určuje název scény, která se má po zavolání metody otevřít. V těle metody přistupujeme k statické metodě *LoadScene* třídy *SceneManager*, která načte danou scénu.

```
public void SwitchScene(string scene)
{
    SceneManager.LoadScene(scene);
}
```

Druhá metoda v této třídě slouží k ukončení hry. *QuitGame* je typu void a nepřijímá žádný parametr. Na rozdíl od předchozí metody, přistupuje tato metoda ke statické metodě *Quit* třídy *Application*. Po zavolání, této metody se hra ukončí.

```
public void QuitGame()
{
      Application.Quit();
}
```

Aby bylo možné s těmito metodami pracovat, tak byl vytvořen prázdný herní objekt, ke kterému se tento skript přiřadil a následně byl z něj vytvořen prefabrikát. Tento prefabrikát je vložen v okně *Hierarchy* v každé scéně. Tento objekt je přiřazen všem tlačítkům sloužícím k přepínání scén.

4.3.2 Pohyb

O pohyb lodí se stará metoda *MoveToDirection* třídy *Engine*. Pomocí metody *GetComponent* vyhledá komponentu *Rigidbody2D* a zavolá její metodu *MovePosition*, která slouží k posouvání herního objektu. Objekt je posunut na novou pozici, která je vypočítaná jako aktuální pozice plus směrový vektor přijatý z parametru metody vynásobený rychlostí pohybu a hodnotou *deltaTime*. *Time.deltaTime* udává dobu mezi aktuálním a přechozím snímkem a při pohybu se používá k tomu, aby rychlost pohybu nebyla závislá na snímkové frekvenci.

```
public void MoveToDirection(Vector2 direction)
{
      GetComponent<Rigidbody2D>().MovePosition(transform.position +
      (Vector3)direction * movementSpeed * Time.deltaTime);
}
```

V odvozené třídě *PlayerEngine* je implementováno rozhraní *IUpgradeable*. To stanovuje, že objekty z této třídy vytvořené lze vylepšovat. O vylepšování se stará metoda *Upgrade*, která po zavolání zvýší atribut *Level* o jedna, atribut *UpgradePrice* se zdvojnásobí, zvýší hodnotu děděného atributu *movementSpeed*, a nakonec aktualizuje text zobrazující stupeň a cenu vylepšení.

```
public interface IUpgradeable
{
      void Upgrade();
      int Level { get; }
      int UpgradePrice { get; }
}
public void Upgrade()
{
      Level++;
      UpgradePrice *= 2;
      movementSpeed *= 1.1f;
      SetMovementSpeedText();
}
private void SetMovementSpeedText()
{
      movementSpeedText.text = "Engine lvl: " + Level + " (" + (Up-
      gradePrice) + "$" + ")";
}
```

4.3.3 Střelba

Střílení má na starost metoda *Fire* třídy *Weapon*, která vytváří projektily pomocí metody *Instantiate*. První parametr udává, jaký objekt má vytvořit. Zde se jedná o prefabrikát projektilu, který je uložen v jednom z atributů. Druhý a třetí parametr říká na jaké pozici a s jakou rotací má být objekt vytvořen. V tomto případě se projektil vytvoří na pozici herního objektu s tímto skriptem a s jeho rotací.

```
public virtual void Fire()
{
	if (CanFire())
	{
		SetNextFireTime();
		Instantiate(projectilePrefab,transform.position,
		transform.rotation);
	}
}
```

Předtím, než zbraň vytvoří projektil, se ještě zkontroluje pomocí metody *CanFire* zda již uplynula dostatečná doba od předchozího výstřelu.

```
protected bool CanFire()
{
	return Time.time >= nextFireTime;
}
```

Pokud ano, tak se nastaví čas, kdy bude zbraň znovu schopna vystřelit metodou *SetNextFireTime*. Tento čas příštího výstřelu je uložen v atributu *nextFireTime* a spočítá se jako aktuální čas plus minimální doba mezi výstřely.

```
protected void SetNextFireTime()
{
	nextFireTime = Time.time + GetFireRate();
}
```

Každá zbraň vystřeluje projektily, které jsou tvořeny prefabrikáty. Všechny tyto prefabrikáty mají komponenty *Transform*, *Sprite Renderer*, *Rigidbody2D*, *Collider2D* a skript *Projectile*. Třída *Projectile* v metodě *FixedUpdate* pohybuje projektilem skoro stejně jako tomu je v metodě *MoveToDirection* v třídě *Engine*. Jediný rozdíl je v tom, že pohyb projektilu je vždy směrem nahoru s ohledem na rotaci objektu díky metodě *up* třídy *Transform*.

```
private void FixedUpdate()
{
        GetComponent<Rigidbody2D>().MovePosition(transform.position
    + transform.up * speed * Time.deltaTime);
}
```

Projektil se při kolizi s jiným objektem zničí. To má na starost metoda *Hit*, která metodou *Destroy* zničí objekt s tímto skriptem. Metoda *OnCollisionEnter2D* je volána při kolizi objektu s jiným objektem.

```
private void Hit()
{
    Destroy(gameObject);
}
private void OnCollisionEnter2D(Collision2D collision)
{
    Hit();
}
```

Implementace třídy *PlayerWeapon* je podobná jako tomu bylo u třídy *PlayerEngine*. Tato třída implementuje rozhraní *IUpgradeable* a také zobrazuje informace o hráčově zbrani v textovém poli.

```
public void Upgrade()
{
    Level++;
    UpgradePrice *= 2;
    SetFireRate(GetFireRate() * 0.9f);
    SetFireRateText();
}
private void SetFireRateText()
{
    fireRateText.text = "Weapon lvl: " + Level + " (" + (Upgrade-
    Price) + "$" + ")";
}
```

Aby ve hře nebyl jen jeden typ nepřátel lišící se rychlostí střelby, rychlostí pohybu či počtem životů, tak byla vytvořena třída *MultiShotWeapon* dědící z třídy *Weapon*. Nepřítel s touto zbraní vystřeluje několik projektilů naráz. Těchto projektilů bude vždy sudý počet a všechny budou létat do stran. V metodě *Awake* je zavolána metoda, která

v případě, že je atribut *numberOfShots* nastaven na lichou hodnotu přidá k tomuto atributu plus jedna.

```
private void Awake()
{
	SetEvenNumberOfShot();
}
private void SetEvenNumberOfShot()
{
	if (numberOfShots % 2 != 0)
	{
		numberOfShots++;
	}
}
```

Vystřelování více nábojů je ošetřeno v přepsané metodě *Fire*. Pokud zbraň může vystřelit, tak se resetuje hodnota lokální proměnné *angle*, která představuje úhel natočení projektilu, na výchozí hodnotu. Poté se opakuje cyklus pro každou střelu, v němž se hodnota *angle* trochu upraví, aby všechny střeli neletěli stejným směrem. Před koncem cyklu se vytvoří projektil pomocí metody *Instatiate*, u kterého se nastaví rotace vynásobením rotace lodi s hodnotou proměnné *angle* pomocí metody *Euler* třídy *Quaternion*. Metoda *Euler* vrací rotaci kolem os x, y a z.

```
public override void Fire()
{
if (CanFire())
	{
	float angle = 10;
	SetNextFireTime();
		for (int i = 1; i <= numberOfShots; i++)
		{
			if (i % 2 == 0)
			{
				angle *= -1;
			}
			else
			{
				angle = i * 15f;
			}
			Instantiate(projectilePrefab, transform.position,
			transform.rotation * Quaternion.Euler(0,0, an-
			gle));
		}
	}
}
```

4.3.4 Správa poškození

O příjem poškození se stará skript *Health*. Ten kontroluje kolize a pokud dojde ke kolizi, která má způsobit danému objektu poškození, tak zavolá metodu *TakeDamage*. Jedná se o přetíženou metodu, která přijímá jeden nebo žádný parametr. Při zavolání této metody se odečte z proměnné *actualHealth* obsahující počet aktuálních životů jeden život či v případě metody s parametrem zadaný počet životů. Tyto metody jsou také označené jako klíčovým slovem *virtuál*. To je z toho důvodu, že v odvozené třídě *PlayerHealth* jsou tyto metody upravené, aby při utržení poškození aktualizovali text zobrazující počet životů hráče zavoláním metody *SetHealthText*.

```
public virtual void TakeDamage()
{
	actualHealth -= 1;
	if (IsDead())
	{
		Die();
	}
}
public virtual void TakeDamage(int damage)
{
	actualHealth -= damage;
	if (IsDead())
	{
		Die();
	}
}
```

Metoda *OnCollisionEnter2D* kontroluje v třídě *Health*, zda se objekt s tímto skriptem srazil s projektily či s nepřátelskými loděmi. Napřed se metoda pokosí zjistit, zda objekt, s kterým došlo ke srážce je typu projektil. Pokud ano, tak dojde k zavolání metody *TakeDamege* s parametrem přijímajícím sílu projektilu. Poté se podobným způsobem zkontroluje, jestli nedochází ke kolizi s nepřátelskou lodí. V případě, že objekt, s kterým došlo ke srážce obsahuje skript *ShipController* a zároveň mají tyto lodě odlišný *tag*, tak dojde k zavolání bezparametrické metody *TakeDamage*. Takto budou obě lodě dostávat po jednom poškození do té doby, dokud se jedna z nich nezničí.

```
private void OnCollisionEnter2D(Collision2D collision)
{
	Projectile projectile = collisi-
on.gameObject.GetComponent<Projectile>();
	if (projectile)
	{
		TakeDamage(projectile.GetDamage());
	}
	ShipController ship = collisi-
on.gameObject.GetComponent<ShipController>();
	if (ship && tag != collision.gameObject.tag)
	{
		TakeDamage();
	}
}
```

Pokud životy klesnou na nulu, tak metoda *IsDead* v metodě *TakeDamege* vrátí hodnotu *true* a dojde k zavolání metody *Die*, která se stará o zničení objektu. Tato metoda je virtuální, a to z důvodu, že v třídě *EnemyHealth* je přepsaná tak aby před zničením objektu zavolala metodu *DropCoin* třídy *EnemyShipController*.

```
private bool IsDead()
{
	return actualHealth <= 0;
}
protected virtual void Die()
{
	Destroy(gameObject);
}
protected override void Die()
{
	GetComponent<EnemyShipController>().DropCoin();
	base.Die();
}
```

4.3.5 Peníze

Každý peníz je vytvořený z prefabrikátu, který obsahuje komponenty *Rigidbody2D*, *CircleCollider2D* a skript *Coin*. Tento skript je velmi jednoduchý. V metodě *FixedUpdate* vyhledá komponentu *Rigidbody2D* a poté s ní pohybuje podobně, jako tomu bylo u projektilů, ale vždy směrem dolů. Pokud peníz narazí do nějakého objektu, tak dojde v metodě *OnCollisionEnter2D* k jeho zničení.

```
private void FixedUpdate()
{
      GetComponent<Rigidbody2D>().MovePosition(transform.position -
      transform.up * fallSpeed * Time.deltaTime);
}
private void OnCollisionEnter2D(Collision2D collision)
{
      Destroy(gameObject);
}
```

4.3.6 Správa peněz

O sběr peněz, jejich utrácení a vypisování se stará skript a stejně pojmenovaná třída *Wallet*. Metoda *OnCollisionEnter2D* zde kontroluje, zda došlo ke srážce s objektem, který má *tag Coin*. Pokud je podmínka pravdivá, tak se do atributu *money* uloží hodnota peníze a dojde k aktualizaci textu zobrazujícího množství sesbíraných peněz metodou *SetMoneyText*.

```
private void OnCollisionEnter2D(Collision2D collision)
{
      if (collision.gameObject.tag == "Coin")
      {
            money += collisi-
            on.gameObject.GetComponent<Coin>().GetValue();
            SetMoneyText();
      }
}
public void SetMoneyText()
{
      moneyText.text = GetMoney() + "$";
}
```

4.3.7 Ovládání lodí

Ovládání hráčských i nepřátelských lodí vychází z abstraktní třídy *ShipController*. Tato třída obsahuje pouze dvě abstraktní metody, které volá v metodě *Update*. Jedná se o metodu *Move* a *Attack*. Tyto metody slouží ke komunikaci se skripty *Engine* a *Weapon*.

```
public abstract class ShipController : MonoBehaviour
{
	private void Update()
	{
		Move();
		Attack();
	}
	protected abstract void Move();
	protected abstract void Attack();
}
```

Třída pro ovládání hráče se jmenuje *PlayerShipController*. V metodě *Awake* nalezne komponenty typu *PlayerWeapon*, *PlayerEngine* a *Wallet*, a uloží je do svých atributů. Také zde nastaví výchozí text zobrazující ovládání lodi.

```
private void Awake()
{
	weapon = GetComponent<PlayerWeapon>();
	engine = GetComponent<PlayerEngine>();
	wallet = GetComponent<Wallet>();
	controlsText.text = "Controls:\n Movement: " + moveToLeftKey
	+ ", " + moveToRightKey + "\n Shooting: " + attackKey + "\n
	Upgrade fire rate: " + upgradeFireRateKey + "\n Upgrade mo-
	vement speed: " + upgradeMovementSpeedKey;
}
```

Dále jsou zde implementované abstraktní metody z nadtřídy, které kontrolují, zda je stisknuto některé z tlačítek uložených v atributech. Ke kontrole stisknutí tlačítka slouží metoda *GetKey* třídy *Input*.

```
protected override void Move()
{
	if (Input.GetKey(moveToLeftKey))
	{
		engine.MoveToDirection(Vector2.left);
	}
	else if (Input.GetKey(moveToRightKey))
	{
		engine.MoveToDirection(Vector2.right);
	}
}

protected override void Attack()
{
	if (Input.GetKey(attackKey))
	{
		weapon.Fire();
	}
}
```

Jsou zde také metody pro vylepšování. *UpgradeMovementSpeed* a *UpgradeFireRate*. Při zavolání některé z těchto metod dojde k odečtení peněz a k vylepšení daného atributu. O aktualizaci stavu peněz se stará metoda *SpendMoney* třídy *Wallet*.

```
private void UpgradeMovementSpeed()
{
	wallet.SpendMoney(engine.UpgradePrice);
	engine.Upgrade();
}

private void UpgradeFireRate()
{
	wallet.SpendMoney(weapon.UpgradePrice);
	weapon.Upgrade();
}
```

Kontrola stisknutí tlačítka pro vylepšení a zároveň kontrola dostatku peněz pro dané vylepšení probíhá v metodě *FixedUpdate*. Pokud se podmínka vyhodnotí jako pravdivá, tak dojde k zavolání příslušné metody k vylepšení.

```
private void FixedUpdate()
{
      if (Input.GetKeyDown(upgradeMovementSpeedKey) && wall-
      et.GetMoney() >= engine.UpgradePrice)
      {
            UpgradeMovementSpeed();
      }
      if (Input.GetKeyDown(upgradeFireRateKey) && wallet.GetMoney()
      >= weapon.UpgradePrice)
      {
            UpgradeFireRate();
      }
}
```

Ovládání nepřítele je poměrně jednoduché. V metodě *Awake* třídy *EnemyShipController* jsou nalezeny a uloženy do příslušných atributů skripty typu *Weapon* a *Engine*. Metoda Attack je implementována tak, že dochází ke generování náhodného čísla a pokud se toto číslo bude rovnat hodnotě atributu *probabilityOfFire*, tak se zavolá metoda *Fire* třídy *Weapon*. Metoda *Move* je volána konstantně a přijímá parametr určující směr dolu, takže nepřítel se neustále pohybuje tímto směrem.

```
protected override void Attack()
{
      if (Random.value < propabilityOfFire)
      {
            weapon.Fire();
      }
}
protected override void Move()
{
      engine.MoveToDirection(Vector2.down);
}
```

EnemyShipController na rozdíl od třídy *PlayerShipController* obsahuje i metodu *OnCollisionEnter2D*, v které dochází ke kontrole, zda nepřítel narazil do spodní hranice. Pokud ano, tak se vyhledají všechny objekty s *tagem Player* a všem se udělí poškození. Poté dojde ke zničení nepřítele.

```
private void OnCollisionEnter2D(Collision2D collision)
{
	if (collision.gameObject.tag == "Shredder")
	{
		var players = GameOb-
		ject.FindGameObjectsWithTag("Player");
		foreach (var player in players)
		{
			player.GetComponent<Health>().TakeDamage();
		}
		Destroy(gameObject);
	}
}
```

4.3.8 Úrovně a vytváření nepřátel

Jednotlivé úrovně definuje třída *Level*. Tato třída je vnořená privátní třída v třídě *EnemySpawner*. Obsahuje privátní atributy a k nim přístupové metody typu *Getter*. Žádné jiné metody v této třídě nejsou.

```
private class Level
{
	[SerializeField]
	private string name;
	[SerializeField]
	private int enemyCount;
	[SerializeField]
	private float spawnRate;
	[SerializeField]
	private Transform[] enemyPrefabs;
	public string GetName()
	{
		return name;
	}
	public int GetEnemyCount()
	{
		return enemyCount;
	}
	public float GetSpawnRate()
	{
		return spawnRate;
	}
	public Transform[] GetEnemyPrefabs()
	{
		return enemyPrefabs;
	}
}
```

V metodě *Awake* se pomocí metody *GetComponentsInChildren* najdou všechny pozice potomků herního objektu, na kterém je tento skript přiřazen. Tito potomci slouží

k určení pozic pro vytváření nepřátel. Nalezené pozice se uloží do pole *spawnPoints*. Také se zde nastaví text zobrazující aktuální úroveň ve hře. Metoda start nastaví atribut *levelCountdown* na hodnotu atributu *timeBetweenLevels*, který udává, jak dlouho se má počkat, než začne další úroveň.

```
private void Awake()
{
      spawnPoints = GetComponentsInChildren<Transform>();
      SetLevelText();
}
private void SetLevelText()
{
      levelText.GetComponent<Text>().text = le-
vels[nextLevel].GetName();
}
private void Start()
{
      levelCountdown = timeBetweenLevels;
}
```

V této třídě se používá výčtový typ *SpawnState*. Ten je využit k určení toho, zda dochází k vytváření nepřátel, k odpočítávání času mezi úrovněmi nebo zda se čeká, než dojde ke zničení všech nepřátel.

```
private enum SpawnState { Spawning, Waiting, Counting };
```

Pokud atribut *state*, obsahující jednu z hodnot *SpawnState*, je roven hodnotě *Waiting*, tak metoda *Update* zkontroluje, zde jsou naživu ještě nějací nepřátelé metodou *EnemyIsAlive*. Pokud již žádný nepřítel nežije, dojde k zavolání metody *LevelCompleted*. Metoda Update dále kontroluje, zda počítadlo *levelCountdown* je roven či nižší než nula. Pokud ano a zároveň není *state* roven hodnotě *Spawning*, tak se metodou *StartCoroutine* spustí metoda *SpawnLevel*. V opačném případě se *levelCountdown* sníží o dobu trvání posledního snímku.

```
private void Update()
{
	if (state == SpawnState.Waiting)
	{
		if (!EnemyIsAlive())
		{
			LevelCompleted();
		}
		else
		{
			return;
		}
	}
	if (levelCountdown <= 0)
	{
		if (state != SpawnState.Spawning)
		{
			StartCoroutine(SpawnLevel(levels[nextLevel]));
		}
	}
	else
	{
		levelCountdown -= Time.deltaTime;
	}
}
```

Metoda *EnemyIsAlive* po uběhnutí doby dané atributem *searchEnemyCountdown* zkusí najít všechny objekty s *tagem Enemy*. Pokud žádný nenajde, tak vrátí hodnotu *false*. V opačném případě metoda vrátí hodnotu *true*.

```
private bool EnemyIsAlive()
{
	searchEnemyCountdown -= Time.deltaTime;
	if (searchEnemyCountdown <= 0f)
	{
		searchEnemyCountdown = 1f;
		if (GameObject.FindGameObjectWithTag("Enemy") == null)
		{
			return false;
		}
	}
	return true;
}
```

Při zavolání metody *LevelCompleted* se *state* nastaví na *Counting* a resetuje se *levelCountdown*. Poté dojde ke kontrole, zda se jednalo o poslední úroveň či ne. Pokud tato úroveň byla poslední, tak se nastaví atribut *lastLevelFinished* na hodnotu *true*. V opačném případě se počítadlo úrovní *nextLevel* zvýší o jedna. Nakonec dojde k aktualizaci textu úrovně.

```
private void LevelCompleted()
{
	state = SpawnState.Counting;
	levelCountdown = timeBetweenLevels;
	if (nextLevel + 1 > levels.Length - 1)
	{
		lastLevelFinished = true;
	}
	else
	{
		nextLevel++;
	}
	SetLevelText();
}
```

Metoda *StartCoroutine* nám umožňuje spustit metodu typu *IEnumerator*, kterou je možno klíčovým slovem *yield* pozastavit. Toho je využito u metody *SpawnLevel*. Na začátku metody se *state* nastaví na hodnotu *Spawning*. Následně se pomocí cyklu vytvoří všichni nepřátelé. Vytváření nepřátel má na starost metoda *SpawnEnemy*, která na náhodném *spawnPointu* vytvoří náhodného nepřítele. Po vytvoření nepřítele metoda počká na určitou dobu, než se cyklus zopakuje. Jakmile se vytvoří všichni nepřátelé, tak se *state* nastaví na *Waiting*.

```
private IEnumerator SpawnLevel(Level level)
{
	state = SpawnState.Spawning;
	for (int i = 0; i < level.GetEnemyCount(); i++)
	{
		SpawnEnemy(level.GetEnemyPrefabs()[Random.Range(0, le-
		vel.GetEnemyPrefabs().Length)]);
		yield return new WaitForSeconds(1f / le-
		vel.GetSpawnRate());
	}
	state = SpawnState.Waiting;
	yield break;
}
private void SpawnEnemy(Transform enemy)
{
	Transform spawnPoint = spawnPoints[Random.Range(0, spa-
wnPoints.Length)];
	Instantiate(enemy, spawnPoint.position, ene-
my.transform.rotation);
}
```

4.3.9 Konec hry

O výhře či prohře rozhoduje třída *GameManager*. Tato třída je velmi jednoduchá. V metodě *Awake* si najde a uloží do příslušných atributů objekty s přiřazeným skriptem *SceneSwitcher* a *EnemySpawner*. Poté v metodě *Update* kontroluje, zda existuje ve scéně objekt typu *PlayerShipController*. V případě, že žádný taková objekt nenajde, tak metodou *SwitchScene* třídy *SceneSwitcher* přepne aktuální scénu na scénu informující o prohře. Dále kontroluje, zda se atribut *lastLevelFinished* rovná hodnotě *true*. Pokud ano, tak se scéna přepne na scénu informující o výhře.

```
private void Awake()
{
      sceneSwitcher = FindObjectOfType<SceneSwitcher>();
      enemySpawner = FindObjectOfType<EnemySpawner>();
}
private void Update()
{
      if (FindObjectOfType<PlayerShipController>() == null)
      {
            sceneSwitcher.SwitchScene("03b LoseScene");
      }
      if (enemySpawner.GetLastLevelFinished())
      {
            sceneSwitcher.SwitchScene("03a WinScene");
      }
}
```

4.4 Závěr

Cílem této práce bylo vytvořit počítačovou hru v herním enginu Unity pomocí objektově orientovaného přístupu v programovacím jazyku C#. V teoretické části práce jsou shrnuty všechny použité technologie. Napřed byl stručně popsán herní engine Unity a jeho grafické prostředí. Poté byl popsán základní stavební kámen všech her vytvářených v Unity a to herní objekt. Následoval popis nejběžnějších komponentů, skriptů a dalších užitečných prvků jako jsou prefabrikáty či vrstvy. Poté byl stručně popsán programovací jazyk C#. Konec teoretické části práce se zaměřil na seznámení se základními koncepty objektově orientovaného programování.

Na začátku praktické části práce došlo k provedení analýzy. Během analýzy byly definovány požadavky na vytvářenou hru a následně navrhnuty různé způsoby řešení. Po dokončení analýzy následovala část návrhu. Tato část byla zaměřena na návrh všech potřebných tříd. Návrh tříd byl proveden v modelovacím jazyce UML. Ve finální části práce byly popsány vybrané části kódu.

Výsledkem této práce je hra, kterou lze hrát v jednom i ve dvou hráčích. Hráči se mohou pohybovat, střílet i vylepšovat svou loď. Ve hře jsou dva typy nepřátel, kteří se liší počtem vystřelovaných projektilů. Systém úrovní také funguje správně. Všechny definované požadavky na hru byly splněny a hra jako taková je plně funkční, proto si myslím, že zadaného cíle práce bylo úspěšně dosaženo. Tuto hru lze také snadno rozšířit například o další úrovně či jiné typy nepřátel.

5 Seznam použitých zdrojů

1. Build once, deploy anywhere. *UNITY: Build once, deploy anywhere* [online]. San Francisco: UNITY, 2000, 2018 [cit. 2019-03-10]. Dostupné z: https://unity3d.com/unity/features/multiplatform
2. Unity store. *Unity store* [online]. San Francisco: UNITY, 2016, 2016 [cit. 2019-03-10]. Dostupné z: https://store.unity.com/
3. A feature-rich and highly flexible editor. *Unity* [online]. San Francisco: UNITY, 2000 [cit. 2019-03-10]. Dostupné z: https://unity3d.com/unity/editor
4. The Project Window. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/2017.3/Documentation/Manual/ProjectView.html
5. The Inspector window. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/2017.3/Documentation/Manual/UsingTheInspector.html
6. SerializeField. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/ScriptReference/SerializeField.html
7. Scene view Control Bar. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/2017.3/Documentation/Manual/ViewModes.html
8. The Game view. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/GameView.html
9. The Hierarchy window. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/2017.3/Documentation/Manual/Hierarchy.html
10. GameObject. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/class-GameObject.html
11. Transform. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/ScriptReference/Transform.html

12. Rigidbody 2D. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/class-Rigidbody2D.html

13. Colliders. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/CollidersOverview.html

14. Creating and Using Scripts [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

15. SHEKHAR, Gyanendu. Difference between Update, FixedUpdate and LateUpdate: Unity Tutorial. *Gyanendu Shekhar's Blog* [online]. INDIA: Gyanendu Shekhar's Blog, 2016, 2018 [cit. 2019-03-10]. Dostupné z: http://gyanendushekhar.com/2018/07/15/update-fixedupdate-lateupdate-unity-tutorial/

16. Prefabs. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/Prefabs.html)

17. Tags. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/class-TagManager.html#Tags

18. Layers. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/Layers.html

19. Úvod do jazyka C# a rozhraní .NET Framework. *Microsoft* [online]. Washington: Microsoft, 1996, 2015 [cit. 2019-03-10]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

20. Objektově orientované paradigma. *Lucka Žolta* [online]. Ostrava: Joomla!®, 2015, 2015 [cit. 2019-03-10]. Dostupné z: http://lucie.zolta.cz/index.php/softwarove-inzenyrstvi/141-objektove-orientovane-paradigma

21. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 15. Microsoft (Artax). ISBN 978-80-87017-02-9.

22. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 17. Microsoft (Artax). ISBN 978-80-87017-02-9.

23. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 15- 16. Microsoft (Artax). ISBN 978-80-87017-02-9.

24. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 16. Microsoft (Artax). ISBN 978-80-87017-02-9.

25. ČADA, Ondřej. Zapouzření. ČADA, Ondřej. *Objektové programování: naučte se pravidla objektového myšlení*. 1. Praha: Grada, 2009, s. 33. Průvodce (Grada). ISBN 978-80-247-2745-5.

26. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 21. Microsoft (Artax). ISBN 978-80-87017-02-9.

27. Object-oriented programming concepts: Objects and classes. *Adobe* [online]. Kalifornie: Adobe, 1996, 2015 [cit. 2019-03-10]. Dostupné z: https://www.adobe.com/devnet/actionscript/learning/oop-concepts/objects-and-classes.html

28. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 21. Microsoft (Artax). ISBN 978-80-87017-02-9.

29. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 22. Microsoft (Artax). ISBN 978-80-87017-02-9.

30. JANSSEN, Thorben. OOP Concepts for Beginners: What is Polymorphism. *Stackify* [online]. Kansas: Stackify, 2011, 2017 [cit. 2019-03-10]. Dostupné z: https://stackify.com/oop-concept-polymorphism/

31. HANÁK, Jan. Objektovo orientované programovanie. HANÁK, Jan. *Objektovo orientované programovanie v jazyku C# 3.0: (príručka pre vývojárov, programátorov a softvérových expertov)*. 1. Brno: Artax, 2008, s. 21. Microsoft (Artax). ISBN 978-80-87017-02-9.

32. Objektově orientované paradigma. *Lucka Žolta* [online]. Ostrava: Joomla!®, 2015, 2015 [cit. 2019-03-10]. Dostupné z: http://lucie.zolta.cz/index.php/softwarove-inzenyrstvi/141-objektove-orientovane-paradigma

33. The Hidden Treasures of Object Composition. *Medium* [online]. Nebraska: A Medium Corporation, 2012, 2017 [cit. 2019-03-15]. Dostupné z: https://medium.com/javascript-scene/the-hidden-treasures-of-object-composition-60cd89480381?fbclid=IwAR02CaEyBbNkFjza4ZCFQ36PMmL77hs83x2QPG67ECLejLooTsz6Cj2G3ks

6 Seznam obrázků

34. The Project Window. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/2017.3/Documentation/Manual/ProjectView.html
35. Console window. In: *Unity Documentation* [online]. San Francisco: Unity, 2014 [cit. 2019-03-15]. Dostupné z: https://docs.unity3d.com/Manual/Console.html
36. *Inspector window*. In: *Unity Documentation* [online]. San Francisco: Unity, 2014 [cit. 2019-03-15]. Dostupné z: https://docs.unity3d.com/Manual/UsingTheInspector.html
37. *Scene view*. In: *Unity Documentation* [online]. San Francisco: Unity, 2014 [cit. 2019-03-15]. Dostupné z: https://docs.unity3d.com/Manual/UsingTheSceneView.html
38. The Game view. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/GameView.html
39. The Hierarchy window. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/2017.3/Documentation/Manual/Hierarchy.html
40. GameObject. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/class-GameObject.html
41. Rigidbody 2D. *Unity documentation* [online]. San Francisco: UNITY, 2014 [cit. 2019-03-10]. Dostupné z: https://docs.unity3d.com/Manual/class-Rigidbody2D.html
42. Architektura .NET Framework. *Microsoft* [online]. Washington: Microsoft, 1996, 2015 [cit. 2019-03-10]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

7 Přílohy

7.1 Projekt v Unity

7.2 Spustitelná hra

7.3 UML diagramy